

目 录

出版者的话
专家指导委员会
译者序
前言

第1章 计算机图形学综述	1
1.1 引言	1
1.2 计算机图形学编程技术的发展	2
1.2.1 硬件层	3
1.2.2 操作系统层的支持	7
1.2.3 GKS 和PHIGS	9
1.2.4 OpenGL	11
1.2.5 Java	13
1.3 Java编程语言	14
1.4 Java 2D	18
1.5 Java 3D	20
1.6 相关领域	23
1.7 参考资料	23
第2章 2D图形学：基础	27
2.1 引言	27
2.2 2D图形绘制过程	27
2.3 2D几何模型与坐标系	28
2.4 Graphics2D 类	29
2.5 绘图方程	33
2.6 几何模型	35
2.6.1 形状	35
2.6.2 实例	37
2.7 构造区域几何模型	43
2.8 一般路径	45
第3章 2D图形学：绘制细节	51
3.1 引言	51
3.2 颜色和涂色	52
3.2.1 颜色空间	52
3.2.2 颜色	52
3.2.3 涂色	56
3.3 笔划	58

3.4 仿射变换	61
3.5 复合变换	68
3.6 透明度和合成规则	71
3.7 裁剪	74
3.8 文本和字体	76
第4章 2D图形：高级话题（可选）	85
4.1 引言	85
4.2 样条曲线	85
4.3 自定义基元	91
4.4 图像处理	94
4.5 创建分形图像	101
4.6 动画	104
4.7 打印	112
第5章 基本3D图形	118
5.1 引言	118
5.2 3D绘制过程	118
5.3 Java 3D API概述	120
5.3.1 一个简单示例	120
5.3.2 安装Java 3D	122
5.4 Java 3D场景图	123
5.5 超结构	125
5.6 节点	126
5.6.1 组节点	126
5.6.2 叶节点	128
5.7 节点组件	129
5.8 Java 3D程序的结构	130
5.9 背景和边界	134
5.10 场景图编译和能力位	140
第6章 图形内容	148
6.1 引言	148
6.2 点和向量	148
6.3 几何特征	150
6.3.1 类GeometryArray	151
6.3.2 类GeometryStripArray	154
6.3.3 类IndexedGeometryArray	155

6.3.4 法向量	159	9.6 纹理映射	263
6.4 类GeometryInfo	161	9.6.1 创建2D纹理映射	263
6.4.1 使用GeometryInfo类	161	9.6.2 纹理坐标	267
6.4.2 创建多边形网格	165	9.6.3 结合纹理映射与光照	268
6.5 几何基元	169	9.6.4 纹理坐标生成	271
6.6 字体和文本	172	第10章 行为和交互	279
6.7 外观和属性	172	10.1 引言	279
第7章 几何变换	183	10.2 行为	279
7.1 引言	183	10.3 交互	286
7.2 3D仿射变换	183	10.3.1 鼠标行为	287
7.2.1 变换矩阵	184	10.3.2 键盘行为	291
7.2.2 类Transform3D	188	10.3.3 视图平台行为	293
7.2.3 旋转	190	10.4 行为和拾取	297
7.3 场景图的变换	198	10.4.1 拾取和鼠标行为	297
7.4 复合变换	201	10.4.2 数据可视化	300
7.5 用变换构造几何体	206	第11章 动画	307
7.5.1 拉伸	206	11.1 引言	307
7.5.2 旋转	208	11.2 Alpha对象	307
7.5.3 变换和共享分支的实例	211	11.3 插值器	311
第8章 视图	218	11.4 变形	322
8.1 引言	218	11.5 细节层次	327
8.2 投影	219	11.6 公告板	331
8.3 视图的定义	220	第12章 其他3D主题	338
8.4 Java 3D的视图模型	222	12.1 引言	338
8.4.1 Java 3D视图配置	223	12.2 3D曲线	338
8.4.2 兼容模式	223	12.3 曲面	342
8.4.3 SimpleUniverse中的视图设置	227	12.3.1 Bézier曲面	342
8.4.4 建立自己的视图	230	12.3.2 犹他茶壶	346
8.5 拾取	234	12.4 声音	349
8.6 头部跟踪	239	12.5 阴影	352
第9章 光照与纹理	248	12.6 几何变化	357
9.1 引言	248	12.7 离屏绘制	363
9.2 光源	249	12.8 3D纹理	367
9.3 光照模型	254	附录A 计算机图形学的数学背景	376
9.4 Java 3D 光照模型	255	附录B 用AWT和Swing进行GUI编程	394
9.5 大气衰减和景深效果处理	259	索引	404

第1章 计算机图形学综述

学习目标

- 了解计算机图形学的基本目标及其适用范围。
- 明确计算机图形学的应用情况。
- 理解2D和3D图形学系统的基本结构。
- 清楚图形编程环境的发展历程。
- 熟悉常用图形应用程序接口。
- 弄清Java语言与Java 2D和Java 3D包所扮演的角色。
- 知晓与计算机图形学相关的领域。

1

1.1 引言

计算机图形学研究关于计算机图形对象的建模、处理与绘制等方面的理论和技术。其基本目标是：构建图形对象的虚拟世界，并按特定视角将虚拟模型的场景在图形设备上绘制出来，如图1-1所示。

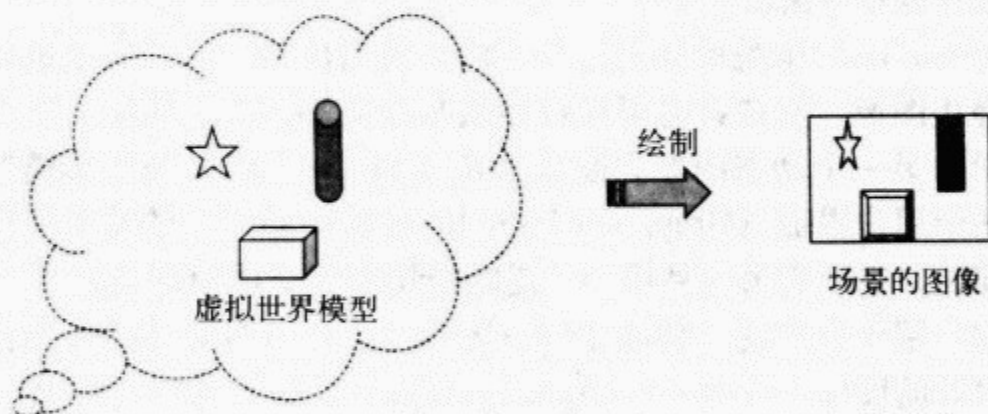


图1-1 计算机图形学的主要任务：构建虚拟世界模型和绘制场景

图形系统一般由建模组件（modeler）和绘制组件（renderer）两大部分组成，建模组件负责虚拟世界模型的构建，绘制组件完成场景的绘制。“保留模式”系统保持图形对象的永久模型，其建模功能是显式的；“立即模式”系统即时地绘制图形对象，但其模型是瞬时的。尽管在某些系统中未必有这样清晰的区分，但这种“建模-绘制”范型（paradigm）的视角对学习图形系统是非常方便的。

一般地说，图形对象的建模既可以在2D空间中进行，也可以在3D空间中进行。通常，将容纳所有图形对象的公共建模空间称为世界空间（world space）。作为图形系统的主要输出内容，世界空间中绘制的场景通常是2D的，因而，2D图形学和3D图形学所涉及的技术相差甚远。由于3D图形学问题要复杂得多，因此，常将2D图形学和3D图形学作为分离的两种课题。

在世界空间中，建模的图形对象通常是诸如线和面之类的几何实体，但像灯光、文字和图像等其他特殊对象也在建模之列。图形对象可以带有许多特征和属性，如：颜色、透明度和纹理等。

几何对象建模可以采用多种不同的数学表示。直线段和简单多边形网格是一种简单而紧凑

的表示方法。它们仅需要存储各种结构形体的顶点，这是很容易实现的。更为复杂的表示形式包括样条曲线和曲面等，它们的表现力强，只需要存储相当少的控点即可表示对象。

将几何变换作用于对象，可将其放置到虚拟空间中的适当位置。这类变换称为对象变换(object transformation)。几何变换也可用来生成视图，这种变换称为视图变换(viewing transformation)。仿射变换(affine transform)是一类非常有用的几何变换，常见的包括平移变换、旋转变换、缩放变换和反射变换等。投影变换(projective transformation)则是更通用的变换，主要用于3D视图生成。

2

视图(view)是从特定视角来观察虚拟世界中模型的结果。2D视图的生成过程(即2D观察)比较简单，其变换通常与对象变换不加区分，只是在变换中要用到诸如合成规则及裁剪路径之类的绘制特征。3D观察要复杂得多，它与眼睛或摄像机的工作方式类似，3D视图的生成过程是将3D对象映射到2D平面的投影过程。诸如投影方式、观察位置、方向和视野等许多参数，都会影响3D图形的绘制。

为获得具真实感的虚拟世界绘制效果，有大量绘制问题需要处理。物体的相对位置需要在绘制的图像中正确地反映出来。例如，一个物体可能被遮挡在另一个物体后面，而被遮挡部分不需要在图像中显示出来；各种特征的光源也需要考虑；物体的材质特性也会对外观产生影响。大多数解决这类问题的方法的计算开销都很大。

硬件设备(hardware device)的性能和特征对图形系统有很大的影响。最常用的图形绘制结果显示输出设备是视频监视器和打印机，其他输出设备包括绘图仪和全息投影仪等。诸如麦克风、鼠标、操纵杆和手写板等输入设备，已经得到广泛应用，而像六自由度跟踪器这样更复杂的输入设备和传感器也不少见。

计算机动画(animation)也是计算机图形学的重要组成部分。动画生成动态的图形内容和绘制结果，而不是静止图像。在诸如电影特技镜头与游戏制作等应用中，动画起着至关重要的作用。计算机图形学的另一个动态图形方面是交互(interaction)，其图形模型需要根据用户输入作相应的改变，图形用户界面(Graphical User Interface, GUI)的基本原理是以用户与图形系统间的交互为基础的。交互的另一种极为广泛的应用实例是视频游戏。

计算机图形学的应用十分广泛。伴随着GUI环境的普及，图形已经成为普通用户程序密不可分的部分。计算机辅助设计(Computer-Aided Design, CAD)和其他工程应用，对图形系统具有很强的依赖性；数据可视化和其他科学应用，同样需要大量地利用图形学；随着诸如CT、PET与MRI等基于计算机的新型仪器的快速发展，医疗系统对计算机图形技术的应用也日趋增加。同时，计算机图形学也是视频游戏及其他娱乐应用中至关重要的组成部分。

传统的计算机图形学必须处理实现细节——用低层算法将线之类的图元扫描转换成像素，确定视图隐藏面，计算曲面上点的颜色值等等。这些低层算法和方法使计算机图形学在技术上显得困难而复杂。本书中，通过使用Java 2D和Java 3D包来避免直接处理很多这样的低层算法细节。如此一来，就可以集中精力去考虑图形建模和绘制方面的大局问题，而不是去关注那些单调乏味的实现细节。

1.2 计算机图形学编程技术的发展

图形编程几乎在计算机体系结构的每个层次都留下了自己的身影。它大致的发展趋势是，从平台相关的低层方法向高层抽象的可移植环境发展。

图1-2给出了处于计算机体系结构中不同层次的图形编程环境实例。在下面的小节中，将讨论不同层次

独立平台(Java 2D和Java 3D)
图形标准(GKS、PHIGS、OpenGL)
操作系统(WIN32、X、Mac OS)
硬件(直接寄存器/视频缓冲区编程)

图1-2 图形编程层次化体系

上图形编程的特点。

1.2.1 硬件层

计算机图形程序依赖于输出设备的图形输出能力。最常用的计算机显示设备包括阴极射线管 (cathode ray tube, CRT) 显示器和液晶显示器 (liquid crystal display, LCD), 它们都是2D光栅设备, 其显示面由离散点的矩形阵列组成。这类显示设备通常由自带处理器和存储器的专门显卡 (图形卡) 驱动。

低层的图形应用通常直接针对图形硬件进行编程。例如, 在运行MS-DOS的典型个人计算机环境下, 大多数图形应用直接访问显示存储器。虽然BIOS和DOS环境对图形功能提供了某种基本的支持, 但是, 对于图形密集型的程序来说, 速度还是显得太慢了。这类程序通常采用汇编语言写成, 并且操作硬件寄存器和视频缓冲区的方法是与机器高度相关的。

程序清单1-1给出了一个低层图形编程的例子, 它是一个汇编语言程序。该程序采用Microsoft Macro Assembler (MMA) 进行汇编, 能够在任意一台带有VGA显卡的IBM PC兼容机上运行。这个程序通过直接写视频缓冲区的内存位置来画圆。圆心在原点的理想圆的方程为:

$$x^2 + y^2 = R^2$$

计算机显示器只能显示离散的像素点。因此, 需要将逼近圆曲线的像素集合计算出来从而显示圆。不过, 实际上只需要计算八分之一圆即可, 其他部分可以通过对称关系得到。如图1-3所示, 算法生成了一系列逼近圆曲线的像素点。现在, 考虑右上角的圆弧部分。从顶端的像素 ($x=0, y=R$) 开始, 算法尝试获得当前像素右侧的下一个像素的位置。它有两个可能的移动方向: “东” 或 “东南”。在这两个候选像素之间, 更接近曲线的那个将被选中。这两种情况也可以通过测试中点 ($x+1, y-0.5$) 来进行选择。如果中点位于圆弧的内侧, 那么选取 “东” 方向的像素, 反之, 选取 “东南” 方向的像素。中点测试可以通过圆的方程来完成。通过使用一些差分变量, 实际的测试过程可进一步简化为只需进行整数加法运算。

程序清单1-1 Circle.asm

```
1 .model small, stdcall
2 .stack 100h
3 .386
4
5 .data
6 saveMode BYTE ? ; 保存视频模式
7 xc WORD ? ; 圆心 x
8 yc WORD ? ; 圆心 y
9 x SWORD ? ; x 坐标
10 y SWORD ? ; y 坐标
11 dE SWORD ? ; “东” 向增量
12 dSE SWORD ? ; “东南” 向增量
13 w WORD 320 ; 屏幕宽度
14
15 .code
16 main PROC
17     mov ax, @data
18     mov ds, ax
19
20     ; 设置视频模式为 320 × 200
21     mov ah, 0Fh ; 获得当前视频模式
22     int 10h
```


4

```
23  mov  saveMode,al ; 保存模式
24
25  mov  ah,0 ; 设置新的视频模式
26  mov  al,13h ; 模式 13h
27  int  10h
28
29  push 0A000h ; 视频段地址
30  pop  es      ; ES = A000h (视频段)
31
32  ;设置背景
33  mov  dx,3c8h ; 视频调色板端口 (3C8h)
34  mov  al,0 ; 设置调色板索引
35  out  dx,al
36
37  ;将屏幕背景颜色设置为深蓝色
38  mov  dx,3c9h ; 端口地址为 3C9h
39  mov  al,0 ; 红色
40  out  dx,al
41  mov  al,0 ; 绿色
42  out  dx,al
43  mov  al,32 ; 蓝色 (32/63)
44  out  dx,al
45
46  ; 画圆
47  ; 将索引1位置的颜色改为黄色(63,63,0)
48  mov  dx,3c8h ; 视频调色板端口 (3C8h)
49  mov  al,1 ; 设置调色板端口索引 1
50  out  dx,al
51
52  mov  dx,3c9h ; 端口地址为 3C9h
53  mov  al,63 ; 红色
54  out  dx,al
55  mov  al,63 ; 绿色
56  out  dx,al
57  mov  al,0 ; 蓝色
58  out  dx,al
59
60  mov  xc,160 ; 屏幕中心
61  mov  yc,100
62
63  ; 计算坐标点
64  mov  x, 0
65  mov  y, 50 ; 半径为50
66  mov  bx, -49 ; 1-半径
67  mov  dE, 3
68  mov  dSE, -95
69
70  DRAW:
71  call Draw_Pixels ; 绘制8个像素点
72
73  cmp  bx, 0 ; 判断方向是 E 还是 SE
74  jns  MVSE
75
76  add  bx, dE ; 向东运动
77  add  dE, 2
78  add  dSE, 2
```

```
79  inc x
80  jmp NXT
81  MVSE:
82  add bx, dSE ; 向东南运动
83  add dE, 2
84  add dSE, 4
85  inc x
86  dec y
87  NXT:
88  mov cx, x ; 如果 x < y 则继续
89  cmp cx, y
90  jb DRAW
91
92  ; 恢复视频模式
93  mov ah, 10h ; 等候按键
94  int 16h
95  mov ah, 0 ; 重设视频模式
96  mov al, saveMode ; 保存模式
97  int 10h
98
99  .EXIT
100 main ENDP
101
102 ; 通过中心对称绘制8个像素点
103 Draw_Pixels PROC
104 ; 计算像素点的视频缓冲区的偏移量
105 mov ax, yc
106 add ax, y
107 mul w
108 add ax, xc
109 add ax, x
110 mov di, ax
111 mov BYTE PTR es:[di], 1 ; 存储颜色索引
112 ; 水平方向对称的像素点
113 sub di, x
114 sub di, x
115 mov BYTE PTR es:[di], 1 ; 存储颜色索引
116 ; 垂直方向对称的像素点
117 mov ax, yc
118 sub ax, y
119 mul w
120 add ax, xc
121 add ax, x
122 mov di, ax
123 mov BYTE PTR es:[di], 1 ; 存储颜色索引
124 ; 水平方向的像素点
125 sub di, x
126 sub di, x
127 mov BYTE PTR es:[di], 1 ; 存储颜色索引
128 ; 变换 x, y 得到其他4个像素点
129 mov ax, yc
130 add ax, x
131 mul w
132 add ax, xc
133 add ax, y
134 mov di, ax
```



```

135  mov BYTE PTR es:[di],1 ; 存储颜色索引
136  sub di, y
137  sub di, y
138  mov BYTE PTR es:[di],1 ; 存储颜色索引
139  mov ax, yc
140  sub ax, x
141  mul w
142  add ax, xc
143  add ax, y
144  mov di, ax
145  mov BYTE PTR es:[di],1 ; 存储颜色索引
146  sub di, y
147  sub di, y
148  mov BYTE PTR es:[di],1 ; 存储颜色索引
149
150  ret
151  Draw_Pixels ENDP
152
153  END main

```

6

程序首先保存当前的视频模式，用BIOS的10h 中断（第27行）切换到模式13h下。视频模式13h是一种简单易用的图形模式，该模式的像素分辨率为 320×200 ，可显示256种颜色。每个像素颜色用视频缓冲区中的一个字节值来表示，缓冲区的起始段地址是A000h。每个字节的值是一个颜色的索引，代表颜色表中给定的一种颜色。由于这种模式的屏幕长宽比与标准显示器不匹配，因此在显示时可能会出现垂直方向的拉伸，使圆实际看起来像椭圆。

程序通过写端口地址为3c9h的寄存器（第38~44行），将屏幕背景颜色设置为深蓝色。圆弧的颜色则设置为黄色（第48~58行）。

圆位于屏幕中心，半径是50，圆心用变量(xc, yc)来定义。从标号DRAW（第70行）开始的循环，用来计算和绘制逼近圆弧的像素。变量(x, y)表示当前像素的坐标，变量dE和dSE表示用于确定下一个移动步的差值。循环通过调用过程Draw-Pixels来绘制当前的像素及其他7个对称像素，同时，它决定如何移动来得到下一个像素并更新相应的变量值。当八分圆的全部像素计算完毕时，程序就退出循环。

Draw-Pixels过程（第103行）绘制对应于当前计算结果的八个像素，将颜色索引写到像素在内存中的对应位置。由于像素在视频缓冲区中的存储地址是线性形式的，因此，有必要计算出合适的地址偏移量，偏移量的计算公式为：

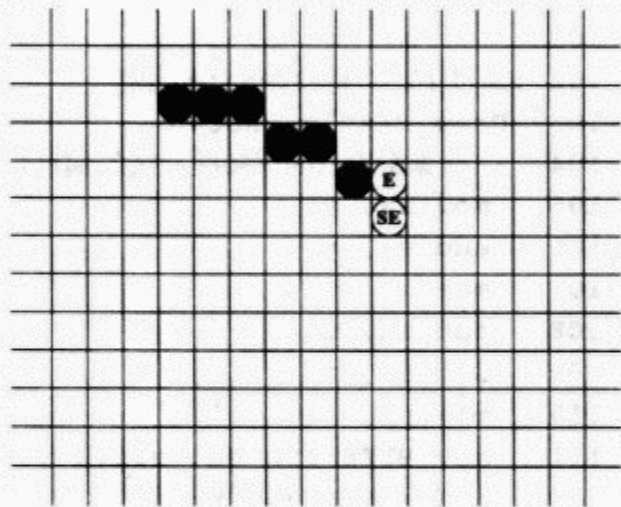


图1-3 确定圆弧上的像素。相对于当前像素，下一个像素只能是位于“东”或“东南”方向的像素之一

7

$$\text{offset} = 320 * x + y$$

视频缓冲区起始于段地址A000h，这个地址值存放于寄存器ES中。当绘制一个像素时，计算像素的位置偏移量，并保存于寄存器DI之中。像素的颜色索引值（在这个例子中是对应黄色的颜色表索引值1）直接写到相应的内存位置。

当圆绘制完毕时，该程序调用中断16h以等待用户按键。一旦接收到按键，就立即恢复视频模式并终止程序运行。

1.2.2 操作系统层的支持

低层的基本图形体系为图形显示编程提供了基础。不过，对于一般的图形应用来说，直接针对视频缓冲区和硬件寄存器进行编程并不是一种有效的方法。正如上一节所介绍的，在硬件层编程需要具备深入的硬件设备知识，这样的编程过程即使对于简单的任务而言，也会显得冗长乏味。在硬件层进行的编程是不可移植的，即使是在同一平台的不同设备上也是如此。

为减轻图形编程的负担，有必要采用高层的编程接口。由于图形问题固有的复杂性，毫无疑问，为应用编程提供一个抽象层将是人们求之不得的，操作系统则是增加这种抽象层次的自然场所。

随着现代计算机系统中图形用户界面（GUI）的发展和广泛应用，各种操作系统对图形编程提供了普遍和广泛的支持。图形应用程序接口（Application Programming Interface, API）在操作系统层面为同一平台上的图形编程提供了统一的接口。通常，硬件的差异由特定硬件的软件驱动程序来处理。软件驱动程序为特定的硬件向操作系统提供标准的接口，应用程序只需要调用操作系统提供的标准图形函数，而不需要处理硬件的特定细节。

WIN32是Windows 9x/ME/NT/2000/XP等32位Windows操作系统的应用程序接口。程序清单1-2给出了在WIN32 下画圆的一个代码实例（如图1-4所示）。这个例子是用C语言编写的一个简单的Windows程序。该程序生成一个标准窗口，并直接调用WIN32 API 在主程序窗口的用户区域画圆。圆心在窗口的中心位置上，当窗口大小变化时，圆的大小会自动调整。

程序清单1-2 WinCircle.c

```
1 #include <windows.h>
2 #include <string.h>
3
4 LRESULT CALLBACK
5 MainWndProc (HWND hwnd, UINT nMsg, WPARAM wParam, LPARAM lParam) {
6     HDC hdc;           /* 用于绘图的设备上下文 */
7     PAINTSTRUCT ps;     /* 绘制过程中所采用的Paint 结构 */
8     RECT rc;           /* 用户区域矩形 */
9     int cx;            /* 中心x坐标 */
10    int cy;             /* 中心y坐标 */
11    int r;              /* 圆的半径 */
12
13    /*消息处理*/
14    switch (nMsg) {
15
16        case WM_DESTROY:
17            /*窗口被取消，结束请求*/
18            PostQuitMessage (0);
19            return 0;
20
21        case WM_PAINT:
22            /* 需要刷新窗口 */
23            hdc = BeginPaint (hwnd, &ps);
24            GetClientRect (hwnd, &rc);
25            /* 计算圆心和半径 */
26            cx = (rc.left + rc.right)/2;
27            cy = (rc.top + rc.bottom)/2;
28            if (rc.bottom - rc.top < rc.right - rc.left)
29                r = (rc.bottom - rc.top) / 2 - 20; /* 半径值 */
30            else
```

```

31     r = (rc.right - rc.left) / 2 - 20; /* 半径值 */
32
33     Ellipse(hdc, cx-r, cy-r, cx+r, cy+r);
34
35     EndPaint (hwnd, &ps); /* 结束绘制 */
36     return 0;
37
38 }
39
40 return DefWindowProc (hwnd, nMsg, wParam, lParam);
41 }
42
43 int WINAPI
44 WinMain (HINSTANCE hInst, HINSTANCE hPrev, LPSTR lpCmd, int nShow){
45     HWND hwndMain;          /* 主窗口柄 */
46     MSG msg;                /* Win32 消息结构 */
47     WNDCLASSEX wndclass;    /* 窗口类结构 */
48     char* szMainWndClass = "WinCircle"; /* 窗口类名 */
49
50     /* 生成一个窗口类 */
51     /* 整个结构的初始值设为0 */
52     memset (&wndclass, 0, sizeof(WNDCLASSEX));
53
54     /* 类名 */
55     wndclass.lpszClassName = szMainWndClass;
56
57     /* 结构的大小 */
58     wndclass.cbSize = sizeof(WNDCLASSEX);
59
60     /* 当调整大小时, 这个类的所有窗口都要进行刷新 */
61     wndclass.style = CS_HREDRAW | CS_VREDRAW;
62
63     /* 这个类的所有窗口都执行MainWndProc 窗口函数 */
64     wndclass.lpfnWndProc = MainWndProc;
65
66     /* 这个类被用在当前的程序实例中 */
67     wndclass.hInstance = hInst;
68
69     /* 采用标准的应用程序图标和箭头光标 */
70     wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
71     wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
72     wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
73
74     /* 背景颜色设为白色 */
75     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
76
77     /* 注册窗口类 */
78     RegisterClassEx (&wndclass);
79
80     /* 用窗口类生成一个窗口 */
81     hwndMain = CreateWindow (
82         szMainWndClass,          /* 类名 */
83         "Circle",                /* 标题 */
84         WS_OVERLAPPEDWINDOW,    /* 字体 */
85         CW_USEDEFAULT,           /* 初始 x值 (用 default型) */
86         CW_USEDEFAULT,           /* 初始 y值 (用 default型) */

```



```

87     CW_USEDEFAULT,          /* 初始 x的大小 (用 default型) */
88     CW_USEDEFAULT,          /* 初始 y的大小 (用 default型) */
89     NULL,                    /* 没有上级窗口 */
90     NULL,                    /* 没有菜单 */
91     hInst,                   /* 程序实例 */
92     NULL                     /* 生成参数 */
93 );
94
95 /* 显示窗口 */
96 ShowWindow (hwndMain, nShow);
97 UpdateWindow (hwndMain); /* 更新窗口 */
98
99 /* 消息循环 */
100 while (GetMessage (&msg, NULL, 0, 0)) {
101     TranslateMessage (&msg);
102     DispatchMessage (&msg);
103 }
104 return msg.wParam;
105 }

```

10

这是使用WIN32 API的一个典型的C语言程序。函数WinMain（第44行）是程序的入口点。该方法创建了一个名为“WinCircle”的窗口类，并为它设置了一组常用的选项。接下来，将该窗口类向Windows系统注册。于是，系统将基于这个注册的类创建、显示并更新其窗口。

同在大多数GUI环境中编程一样，Windows编程遵循事件驱动模型。消息发送给窗口，窗口的消息处理函数通过执行一定的操作来响应消息。函数WinMain最后的循环是一个标准的消息循环，对接收到的消息进行分配。

MainWndProc（第5行）函数是一个负责处理发送给窗口的消息事件的窗口过程。它是在窗口类定义中说明的一个回调函数。在这个例子中，该函数处理两类消息：当用户试图关闭窗口时通常会发送的WM_DESTROY消息，以及当系统尝试重新绘制窗口时会发送的WM_PAINT消息。如果接收到WM_DESTROY消息，处理函数将通过发出一个退出消息来终止程序。当接收到WM_PAINT消息时，处理函数将在窗口中绘制一个圆，绘制过程是通过调用BeginPaint函数来获得一个设备上下文而完成的。当宽和高相同时，采用Ellipse函数（第33行）可以绘制圆。EndPaint函数用来结束绘制。圆心和半径是基于窗口用户区域大小计算出来的，而窗口用户区域的大小则通过调用GetClientRect函数来获取。

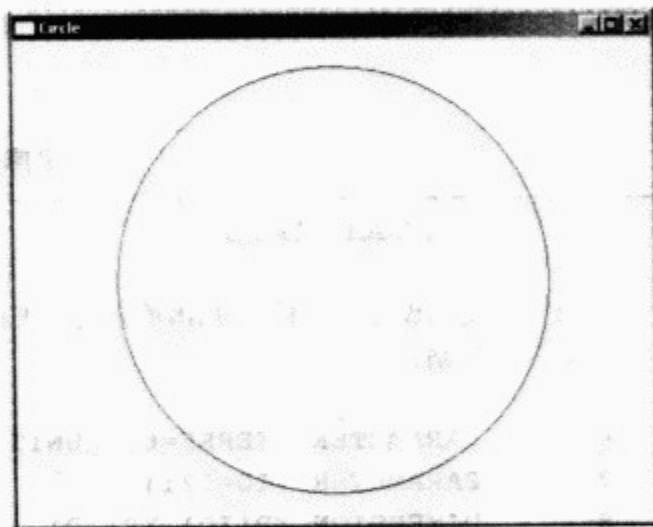


图1-4 用C语言编写的一个WIN32画圆程序

1.2.3 GKS 和PHIGS

相对于硬件层的方法，基于操作系统API的图形编程，在设备无关性和方便性方面前进了一大步。但是，这种图形编程方法依赖于操作系统函数，显然是无法跨平台的。例如，Microsoft Windows 和Mac OS 虽然都是支持图形用户界面的操作系统，但是它们的API是不同的，在系统调用层次并不兼容。

图形编程的标准化接口的优点是显而易见的，图形编程标准提供一个必要的抽象层，以支持对设备和平台的无关性。在计算机图形学发展的短暂历史中，出现了几种著名的图形标准。

图形核心系统 (Graphics Kernel System, GKS) 是第一个有关计算机图形的国际标准。GKS (ISO 7942 1985) 是一种面向2D图形学的标准, 它定义了一组不依赖于计算机平台的基本图形函数。该标准定义了多个层次, 以适应不同硬件系统的不同能力。在某种特定编程语言的GKS的实现中, 必定需要适合该语言的语法定义。语言绑定 (language binding) 被用来定义编程语言中GKS的具体格式。最常见的GKS语言绑定是FORTRAN, 其他还有Pascal和C语言等的语言绑定。

GKS-3D (ISO 8805 1988) 是GKS的扩展, 用于支持3D图形学。GKS和CKS-3D API 主要是用来绘制带有特定属性的单个物体, 它们非常适合于绘制非结构化的静态图元, 但不直接支持比较复杂的图形模型。

程序员的分层交互式图形系统 (Programmer's Hierarchical Interactive Graphics System, PHIGS) (ISO 9592 1991) 是一种类似于GKS的图形标准。虽然它们不是GKS严格意义上的扩展集, 但PHIGS和PHIGS+还是包括了GKS的功能。它们具有额外的功能, 可以对基本图形进行分层的组织, 并进行动态编辑。

程序清单1-3给出了在FORTRAN 语言绑定下进行GKS编程的一个实例。这个简单的FORTRAN程序用GKS的折线基元绘制了一个红色的圆 (如图1-5所示)。圆上的点通过FORTRAN语言提供的高层三角函数计算得到。其计算公式如下:

$$x = x_0 + r \cos \theta$$

$$y = y_0 + r \sin \theta$$

程序清单1-3 circle.f

```

1      PROGRAM CIRCLE
2      C
3      C  定义错误文件, FORTRAN单元数, 工作站类型与
4      C  工作站ID
5      C
6      C  PARAMETER (IERRF=6, LUNIT=2, IWTYPER=1, IWKID=1)
7      C  PARAMETER (ID=121)
8      C  DIMENSION XP(ID), YP(ID)
9      C
10     C  打开GKS, 打开和激活工作站
11     C
12     C  CALL GOPKS (IERRF, IDUM)
13     C  CALL GOPWK (IWKID, LUNIT, IWTYPER)
14     C  CALL GACWK (IWKID)
15     C
16     C  定义颜色
17     C
18     C  CALL GSCR(IWKID, 0, 1.0, 1.0, 1.0)
19     C  CALL GSCR(IWKID, 1, 1.0, 0.0, 0.0)
20     C
21     C  画一个圆
22     C
23     C  X0 = .5
24     C  Y0 = .5
25     C  R  = .3
26     C  JL = 120
27     C  RADINC = 2.*3.1415926/REAL(JL)
28     C  DO 10 J=1, JL+1
29     C  X = X0+R*COS(REAL(J)*RADINC)

```



```

30      Y = Y0+R*SIN(REAL(J)*RADINC)
31      XP(J) = X
32      YP(J) = Y
33  10  CONTINUE
34      CALL GSPLI(1)
35      CALL GSPLCI(1)
36      CALL GPL(JL+1,XP,YP)
37  C
38  C  解除和关闭工作站, 关闭GKS
39  C
40      CALL GDAWK (IWKID)
41      CALL GCLWK (IWKID)
42      CALL GCLKS
43  C
44      STOP
45      END

```

12

GKS函数通常以“G”作为函数名前缀。对函数GOPKS、GOPWK和GACWK的调用（第12~14行）设置了GKS环境，对GSCR的调用定义颜色的索引。圆是由120个点的折线段来定义的，这些点通过直接调用COS和SIN函数进行计算，另外增加一点用来封闭曲线。对GPL的调用（第36行）是用来绘制折线的。

该程序是在Linux操作系统下用NCAR图形包编译的。

1.2.4 OpenGL

OpenGL是从Silicon Graphics公司的GL (Graphics Library, 图形库) 衍生而来的一种流行的2D/3D图形应用程序接口。GL是在SGI公司的图形工作站上得到成功应用的一种图形程序接口。OpenGL则被设计为一种开发商中立的开放工业标准，OpenGL差不多可用于所有的计算机平台。实际上，许多硬件厂商在他们的显卡和设备上提供了OpenGL接口。OpenGL拥有200个以上的函数，提供了比早期的GKS标准功能强大得多的图形API。

OpenGL是一种相对低层的API，提供的是一种面向过程的接口。OpenGL可以像GKS一样，有不同的语言绑定形式。已经有一种官方的FORTRAN语言绑定可用，而目前有一种Java语言的绑定正在开发中。不过，OpenGL仍然是深深植根于C语言的，它的C语言绑定是最常用的一种绑定。

OpenGL由两个库组成：GL和GLU (OpenGL Utility Library)。GL库包含了提供基本图形特征的核心函数，GLU库包含了高级别的基于GL函数的应用函数。OpenGL本身没有提供用于构造用户接口的函数。不过，可以通过一个简单的可移植包GLUT (OpenGL Utility Toolkit) 来构造一个完整的图形程序。

程序清单1-4给出了一个简单的OpenGL画圆示例程序（如图1-6所示）。程序用GLUT构建用户接口，用GL和GLU函数实现显示。GL、GLU与GLUT库中的函数名通常用库名作为前缀。

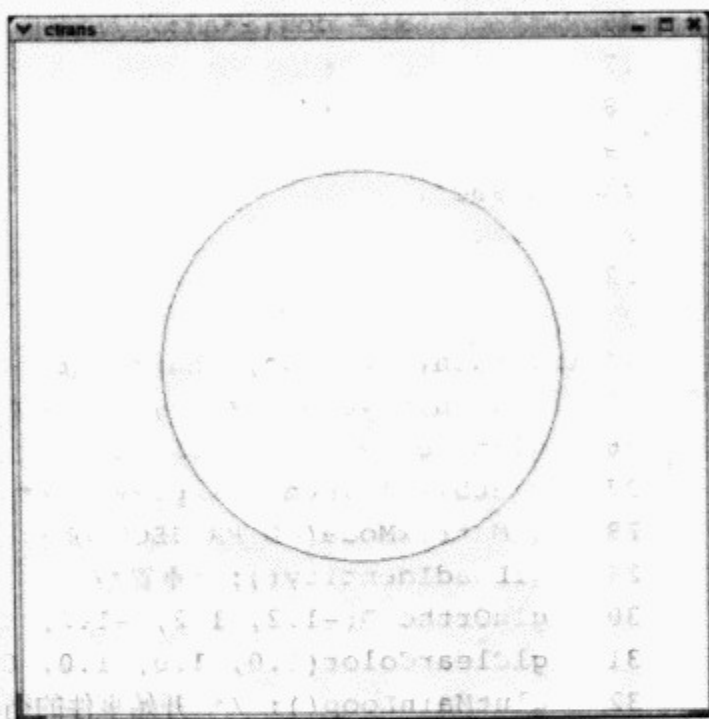


图1-5 用以显示圆的简单的GKS程序

13

13

程序清单1-4 OpenGLCircle.c

```

1 #include <GL/glut.h>
2 #include <math.h>
3
4 void display(void) { /*定义显示函数 */
5     int i;
6     int n = 80;
7     float a = 2*3.1415926535/n;
8     float x;
9     float y;
10
11     glClear(GL_COLOR_BUFFER_BIT); /* 把屏幕上的颜色清空 */
12     glColor3f(1.0,0,0);
13
14     glBegin(GL_LINE_LOOP);
15     for (i = 0; i < n; i++) {
16         x = cos(i*a);
17         y = sin(i*a);
18         glVertex2f(x, y);
19     }
20     glEnd();
21     glFlush();
22 }
23
24 int main(int argc, char** argv) {
25     glutInit(&argc, argv); /* 初始化GLUT */
26     glutCreateWindow("Circle"); /* 创建一个窗口 */
27     glutDisplayFunc(display); /* 设置显示函数 */
28     glMatrixMode(GL_PROJECTION); /*指明接下来的两行代码将影响投影矩阵*/
29     glLoadIdentity();/*重置*/
30     gluOrtho2D(-1.2, 1.2, -1.2, 1.2); /* 显示的投影矩阵, 2D正交投影 */
31     glClearColor(1.0, 1.0, 1.0, 0.0);
32     glutMainLoop(); /* 开始事件的循环 */
33 }

```

14 main函数调用几个GLUT函数来设置显示窗口、显示函数和消息循环。glutInit函数（第25行）用以初始化GLUT。glutCreateWindow函数的作用是创建一个窗口。glutDisplayFunc函数用来设置显示函数，它是个回调函数，用来进行图形绘制。glutMainloop函数（第32行）用来启动事件循环。

显示的投影矩阵设置为一个2D正交投影，这是通过调用几个GL函数和GLU的gluOrtho2D函数来实现的。

display函数（第4行）定义为这个程序的显示函数，它使用GL_LINE_LOOP模式中的顶点序列来绘制圆。顶点通过圆的参数方程进行计算，并通过调用glVertex2f函数进行设置。

当然，OpenGL作为3D应用程序接口，可以做比画一个圆更为复杂的事情。程序清单1-5给出了另一个简单的OpenGL程序例子，该程序绘制一个旋转的3D球体（如图1-7所示）。

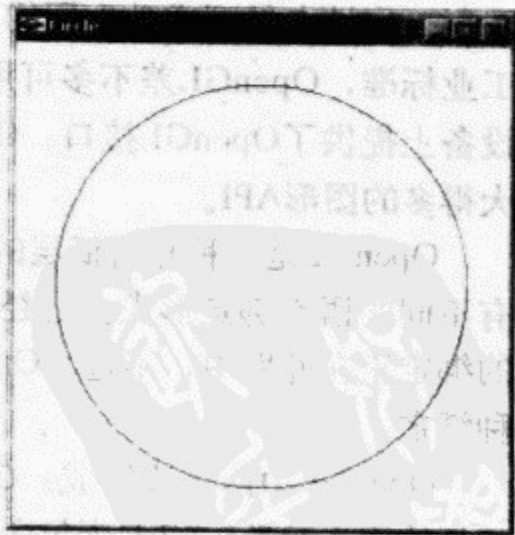


图1-6 用OpenGL 绘制的圆

程序清单1-5 OpenGLSphere.c

```

1 #include <GL/glut.h>
2
3 GLUQuadricObj* sphere;
4
5 void display(void) {
6     glClear(GL_COLOR_BUFFER_BIT); /* 把屏幕上的颜色清空 */
7     glMatrixMode(GL_MODELVIEW); /* 指明任何新的变换将会影响模型观察矩阵 */
8     glRotatef(0.2, 0.0, 0.0, 1.0); /* 进行旋转, 方向为 (0, 0, 1), 角度为 0.2 */
9     gluSphere(sphere, 1.8, 24, 24);
10    glutSwapBuffers(); /* 交换当前窗口使用层的缓存, 将后台缓存内容交换到前台 */
11 }
12
13 void idle(void) {
14     glutPostRedisplay(); /* 显示函数 */
15 }
16
17 int main(int argc, char** argv) {
18     glutInit(&argc, argv);
19     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB); /* 实现双缓冲 */
20     glutCreateWindow("Spinning Sphere");
21     glutDisplayFunc(display); /* 注册当前窗口的显示回调函数 */
22     glMatrixMode(GL_PROJECTION); /* 指明接下来的两行代码将影响投影矩阵 */
23     glLoadIdentity();
24     glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0); /* 投影矩阵, 正交投影 */
25     glClearColor(1.0, 1.0, 1.0, 0.0);
26     glColor3f(1.0, 0.5, 0.5);
27     sphere = gluNewQuadric();
28     gluQuadricDrawStyle(sphere, GLU_LINE);
29     glutIdleFunc(idle);
30     glEnable(GL_CULL_FACE); /* 将不渲染看不见的隐消面 */
31     glCullFace(GL_BACK); /* 剔除背面 */
32     glMatrixMode(GL_MODELVIEW); /* 指明任何新的变换将会影响模型观察矩阵 */
33     glLoadIdentity();
34     gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); /* 定义观测角度 */
35     glutMainLoop();
36 }

```

15

在这个例子中, 通过函数 `glutInitDisplayMode` (第19行) 来实现双缓冲。用于显示的投影矩阵通过GL函数设置为正交投影。函数调用 `gluLookAt` (第34行) 定义观察角度, 眼睛的位置在(1,1,1), 注视的位置在(0,0,0)。通过GLU函数创建球体对象。

在这个例子中, 除了显示回调外, 还加入了一个idle回调来驱动动画。idle函数 (第13行) 调用 `glutPostRedisplay` 函数来请求对显示函数的调用。在显示函数中, 模型的视图矩阵会有小角度的旋转。接下来, 在隐藏的缓冲区内重新绘制球体。最后, 两个缓冲区进行交换, 从而显示最新的绘制结果。

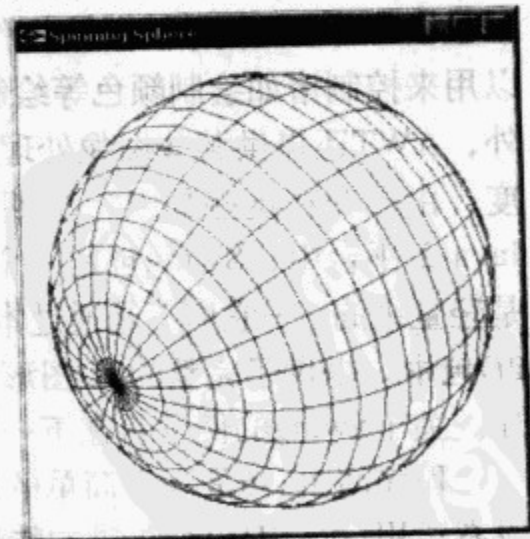


图1-7 用OpenGL绘制3D球体

1.2.5 Java

OpenGL对图形硬件提供了一个标准的和高效率的绘制接口。不过, 随着计算机硬件和软

件技术的高速发展，可以对图形编程提供更高层次的抽象。OpenGL提供了一种类似于C语言的面向过程的抽象，而不是专门针对在面向对象的编程模式中直接进行图形建模而设计的。一种基于面向对象编程（object-oriented programming, OOP）技术的高层图形API（可由OpenGL构建），或许能为应用程序员提供更多的好处。

Java 2D 和Java 3D是与Java编程语言相关的新型图形API。它们作为一种面向对象的高层API，具有高度的可移植性，本书就采用Java 2D和Java 3D作为编写程序的API。Java 3D 通常基于OpenGL之类的其他低层API来实现。图1-8给出了一个典型的图形系统层次结构。

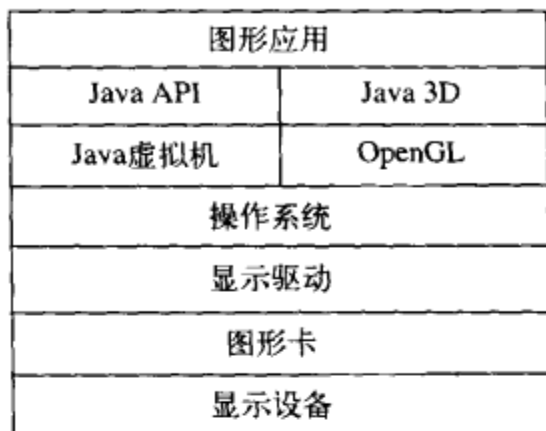


图1-8 图形系统的层次结构

在接下来的三节里，将对Java语言及其图形功能进行概要介绍。

1.3 Java编程语言

Java 是一种功能完备、通用性强的编程语言，可以用来编写健壮的关键任务应用程序。近年来，Java语言十分流行，在广泛的应用中迅速成为首选的编程语言。今天，它不仅用于Web编程，而且用于开发跨服务器、桌面计算机和移动设备等多种平台的独立应用程序。

Java程序被编译成一种平台无关的标准格式，这种格式被称为“字节码”（byte code）。编译得到的字节码不经任何改变，就可以在任意一台具有Java虚拟机（Java Virtual Machine）的机器上运行。这种平台无关的特性，使Java语言成为通过Internet分发应用程序的理想编程语言。

Java语言是为从根本上支持面向对象编程技术（OOP）而进行设计的。一个Java 程序完全由类定义组成，对象的实例化和交互是Java程序的主要行为。

Java语言继承了其前身C程序语言的简单、优雅和高效的特性。同时，Java语言消除了C和C++的一些缺陷和不足。

尽管Java语言本身十分简单，但Java平台提供了一组完整的应用程序接口。Java API覆盖了广泛的任务和应用范围，比如说文件I/O、图形处理、多媒体应用、数据库处理、网络和安全等。

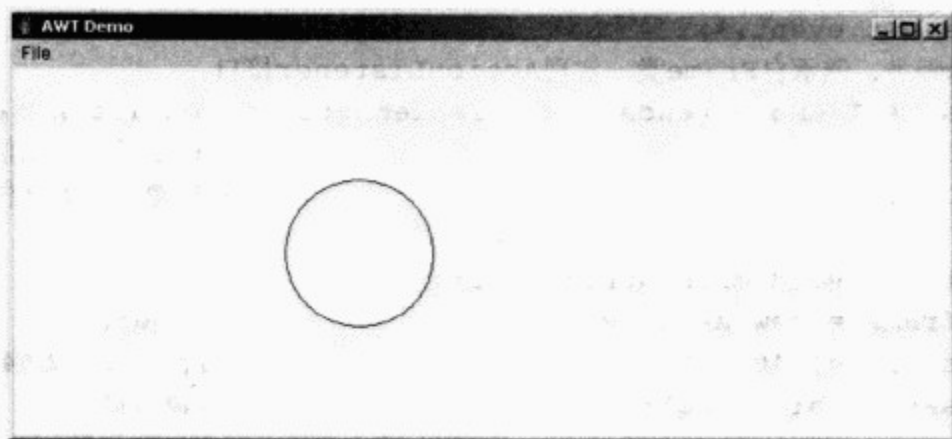
针对GUI编程应用，Java语言提供了两套几乎并行的工具：AWT和Swing。早期的Java版本提供了有限的图形支持。JDK 1.x 版本只包含最小的图形特征集。抽象窗口工具包（Abstract Window Toolkit, AWT）提供对图形用户界面的支持和一些图形绘制功能。AWT的GUI组件是重量级的——它们被映射为操作系统的本地组件。除了创建GUI元素的简单功能外，AWT还可以用来控制诸如绘制颜色等绘制特性，并绘制简单的图形基元，例如直线、矩形和椭圆等。此外，AWT还提供某些图像处理功能。不过，AWT的功能十分有限。例如，它不能控制画线宽度。由于存在这些局限，早期的Java版本不能够对现代计算机图形编程提供足够的支持。在Java 2 平台中，Swing包是一个经过完全重新设计的图形用户界面API库。大多数Swing组件都是轻量级的——它们不是通过本地组件实现的。Java 2对图形的支持也有了很大的增强。在Java 2D包中，提供了完整的2D图形功能。程序清单1-6给出了一个只使用AWT包的简单Java GUI程序。关于Swing的例子将在下一节中给出。

程序清单1-6是一个简单的Java GUI应用程序，它只运用了AWT提供的简单绘制功能，而没有运用Java 2D中更先进的特性。它在一个窗口中绘制一个圆（如图1-9所示）。如果用户在窗口中点击鼠标，这个圆将移到新的位置，圆心就是用户鼠标的点击位置。菜单中增加了一项“Exit”，当用户选择该项时，程序终止。

程序清单1-6 AWTDemo.java

```
1 package chapter1;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 //定义AWTDemo类, 继承自Frame类, 实现ActionListener接口
6 public class AWTDemo extends Frame implements ActionListener{
7     int x = 100; //x坐标变量, 默认值为100
8     int y = 100; //y坐标变量, 默认值为100
9
10    public static void main(String[] args) {
11        Frame frame = new AWTDemo(); //创建主窗口
12        frame.setSize(640, 480); //设置窗口大小为640×480
13        frame.setVisible(true); //设置窗口可见
14    }
15    // AWTDemo类构造函数
16    public AWTDemo() {
17        setTitle("AWT Demo"); //设置窗口标题
18        //开始设置菜单栏
19        MenuBar mb = new MenuBar();
20        setMenuBar(mb);
21        Menu menu = new Menu("File");
22        mb.add(menu);
23        MenuItem mi = new MenuItem("Exit");
24        mi.addActionListener(this);
25        menu.add(mi);
26        //添加窗口事件侦听器, 处理窗口关闭事件
27        WindowListener l = new WindowAdapter() {
28            public void windowClosing(WindowEvent ev) {
29                System.exit(0);
30            }
31        };
32        this.addWindowListener(l);
33        // 添加鼠标事件侦听器, 处理鼠标点击事件
34        MouseListener mouseListener = new MouseAdapter() {
35            public void mouseClicked(MouseEvent ev) {
36                x = ev.getX(); //获得鼠标当前位置的x坐标值
37                y = ev.getY(); //获得鼠标当前位置的y坐标值
38                repaint(); //重绘窗口区域
39            }
40        };
41        addMouseListener(mouseListener);
42    }
43    //重写了paint方法, 用于绘制窗口
44    public void paint(Graphics g) {
45        g.drawOval(x-50, y-50, 100, 100); //在参数确定的区域绘制圆
46    }
47    //事件响应方法
48    public void actionPerformed(ActionEvent ev) {
49        String command = ev.getActionCommand();
50        if ("Exit".equals(command)) { //用户点击了菜单按钮 "Exit"
51            System.exit(0); //退出程序
52        }
53    }
54 }
```


这个程序是一个运用AWT编写的GUI应用程序，它的主窗口包括一个菜单和一个圆。菜单中只包括一个选项“Exit”，当用户选择该选项时关闭窗口。用户点击鼠标时，图形绘制逻辑会作出响应，从而在鼠标的点击位置重新绘制图形。



18

图1-9 用AWT实现的一个简单Java GUI程序

AWTDemo类被定义为Frame（第6行）的一个子类，它定义了程序主窗口。在窗口上的菜单是通过MenuBar、Menu和MenuItem（第19~25行）类的对象来创建的。AWTDemo类实现了ActionListener接口，处理由菜单选择所产生的ActionEvent。接口中定义的actionPerformed方法负责事件的处理。当菜单中“Exit”选项被选中时，程序通过调用方法System.exit(0)而终止。

另外两个事件处理函数，在AWTDemo类的构造函数中定义。WindowListener定义为内部的WindowAdapter（第27~32行）的匿名派生类，它重载在收到窗口关闭事件时终止程序运行的windowClosing方法。另一个侦听器是MouseListener，它是MouseAdapter的派生类（第34~41行），它重写了mouseClicked方法，以处理鼠标点击事件。在mouseClicked方法中，鼠标位置保存在变量x和y中，并调用repaint方法刷新图形，从而把圆移到新的位置。

方法paint（第44行）调用Graphics对象的drawOval方法，绘制一个半径为50的圆，圆心位置由变量x和y的值来确定。

main方法创建和显示一个AWTDemo类的实例，窗口大小被定义为640×480。

Java语言图形编程的另一个选择是使用OpenGL。目前有若干项目正在开发OpenGL的Java语言绑定。JOGL作为JSR231的一种实现形式，对OpenGL进行了Java语言绑定。JOGL提供GL和GLU两个类，来封装GL和GLU库的函数。组件GLCanvas和GLJPanel为OpenGL函数调用提供绘制表面。GLCanvas是重量级组件，能够使用硬件加速功能。GLJPanel是在系统内存中实现的轻量级组件，不能使用硬件加速功能。用JOGL进行编程的典型过程如下所示：

- 1) 通过GLDrawableFactory类生成GLCanvas或GLJPanel的对象。
 - 2) 为画布对象添加GLEvent 侦听器。
 - 3) 事件侦听器的实现通过init、display、reshape和displayChanged四种方法的实现来完成。
- 程序清单1-7给出了与程序清单1-4功能等价的JOGL程序。

程序清单1-7 JOGLDemo.java

```
1 package chapter1;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import net.java.games.jogl.*;
7 //定义AWTDemo类
8 public class JOGLDemo {
```



```
9
10 public static void main(String[] args) {
11     Frame frame = new Frame("JOGL Demo");           //创建主窗口, 设定标题栏
12     GLCapabilities cap = new GLCapabilities();
13     GLCanvas canvas =
14     GLDrawableFactory.getFactory().createGLCanvas(cap);
15     canvas.setSize(300, 300);                        //显示窗口大小被定义为300*300
16     canvas.addGLEventListener(new Renderer());
17     frame.add(canvas);                               //加入GLCanvas对象
18     frame.pack();
19     frame.addWindowListener(new WindowAdapter() {
20         public void windowClosing(WindowEvent e) {
21             System.exit(0);
22         }
23     });
24     frame.show();                                    //设置窗口可见
25 }
26 //定义静态类Renderer
27 static class Renderer implements GLEventListener {
28     private GL gl;
29     private GLU glu;
30     private GLDrawable gldrawable;
31
32     public void init(GLDrawable drawable) {
33         gl = drawable.getGL();
34         glu = drawable.getGLU();
35         this.gldrawable = drawable;
36         gl.glMatrixMode(GL.GL_PROJECTION);           //选择投影矩阵
37         gl.glLoadIdentity();                         //重置当前的模型观察矩阵
38         glu.gluOrtho2D(-1.2, 1.2, -1.2, 1.2);        //初始化观察参数
39         gl.glClearColor(1.0f, 1.0f, 1.0f, 0.0f);    //设置像素清除颜色
40     }
41     //定义显示函数
42     public void display(GLDrawable drawable) {
43         int i;
44         int n = 80;
45         float a = (float)(2*3.1415926535/n);
46         float x;
47         float y;
48
49         gl.glClear(GL.GL_COLOR_BUFFER_BIT);          //清除屏幕像素
50         gl.glColor3f(1.0f, 0, 0);                    //设置画笔颜色
51         gl.glBegin(GL.GL_LINE_LOOP);                 //画圆
52         for (i = 0; i < n; i++) {
53             x = (float)Math.cos(i*a);
54             y = (float)Math.sin(i*a);
55             gl.glVertex2f(x, y);
56         }
57         gl.glEnd();
58         gl.glFlush();                                //强制执行缓冲区命令
59     }
60
61     public void reshape(GLDrawable drawable, int x, int y, int width,
62         int height) {}
63     public void displayChanged(GLDrawable drawable,
```

```

64         boolean modechanged, boolean deviceChanged) {}
65     }
66 }

```

20

JOGL作为一种OpenGL的语言绑定形式，具有和OpenGL相同的优缺点。它能够进行高效率的绘制，但却不能够提供具有Java语言面向对象特征的完备图形建模能力。

1.4 Java 2D

通过引入Swing、Java 2D和Java 3D API，Java 2 平台的图形能力得到了很大的改善。应用程序接口的完善设计，为计算机图形的各种任务提供了全面支持。加上Java程序语言独特的优点，以及Java语言和Java 2D及Java 3D包的组合，使其成为进行图形编程和学习计算机图形学的一种很有吸引力的选择。

早期的Java版本对图形学的支持是十分简单和有限的。Java 2D提供了一组十分完备的功能，可以用来操纵和绘制2D图形。具体的改善包括：

- Java 2D为几何对象定义了一个单独的类层次。
- 绘制过程更加精细。
- 引入了全新的图像处理功能。
- 对于颜色模型、字体、打印和其他图形相关的支持，也有了很大的提高。

Graphics2D类是Graphics类的一个子类，是Java 2D的一个绘制引擎。它提供的方法可以绘制几何体、图像和文本。绘制过程的控制包括选择变换、涂色、线性、合成、裁剪路径和其他属性。

Java 2 平台的Swing组件和Java 2D包，比早期Java平台的图形工具更为高级。本书中Java 2D的例子将尽可能地避免使用旧的AWT组件而使用Swing组件。

程序清单1-8简单地演示了Java 2D的图形功能。它运用了一些Java 2D的高级性能，例如透明度、渐变涂色、变换和字体字形等，这些都是AWT中没有的功能（绘制结果如图1-10所示）。

程序清单1-8 Demo2D.java

```

1 package chapter1;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.font.*;
7 import java.awt.geom.*;
8 //定义Demo2D类，继承自JApplet类，可作为applet运行
9 public class Demo2D extends JApplet {
10     public static void main(String s[]) {
11         JFrame frame = new JFrame(); //创建主窗口
12         frame.setTitle("Java 2D Demo"); //设置标题栏
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //关闭窗口处理
14         JApplet applet = new Demo2D(); //生成Demo2D类实例并初始化
15         applet.init();
16         frame.getContentPane().add(applet); //将Demo2D实例加入到frame内容窗格
17         frame.pack();
18         frame.setVisible(true); //设置窗口可见
19     }
20     //重写初始化函数

```



```

21 public void init() {
22     JPanel panel = new Panel2D();
23     getContentPane().add(panel);
24 }
25 }
26 //定义Panel2D类
27 class Panel2D extends JPanel{
28     public Panel2D() {
29         setPreferredSize(new Dimension(500, 400)); //设置组件首选大小
30         setBackground(Color.white); //设置背景颜色为白色
31     }
32     //重写绘制组件方法
33     public void paintComponent(Graphics g) {
34         super.paintComponent(g);
35         Graphics2D g2 = (Graphics2D)g;
36         //绘制一个椭圆
37         Shape ellipse = new Ellipse2D.Double(150, 100, 200, 200);
38         GradientPaint paint =
39             new GradientPaint(100,100, Color.white, 400, 400, Color.gray);
40         g2.setPaint(paint); //设置椭圆渐变涂色效果
41         g2.fill(ellipse); //填充椭圆
42         //设置透明度
43         AlphaCompsite ac =
44             AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.4f);
45         g2.setComposite(ac);
46         g2.setColor(Color.blue);
47         //绘制透明文本
48         Font font = new Font("Serif", Font.BOLD, 120);
49         g2.setFont(font);
50         g2.drawString("Java", 120, 200);
51         //获得文本字体的轮廓线
52         FontRenderContext frc = g2.getFontRenderContext();
53         GlyphVector gv = font.createGlyphVector(frc, "2D");
54         Shape glyph = gv.getOutline(150,300);
55         //绘制可旋转的字体
56         g2.rotate(Math.PI/6, 200, 300);
57         g2.fill(glyph);
58     }
59 }

```

21

Swing组件的类名一般都带有前缀“J”。Panel2D类（第27行）是JPanel类的扩展，它重写paintComponent方法（第33行）。方法中的Graphics类型参数被强制转换成Graphics2D类型，以便利用Java 2D的扩展功能。采用渐变涂色法来绘制圆，其绘制颜色随着位置的变化而变化。然后，通过对合成规则的设置，可以达到某种程度的透明效果。获取字符文本串“2D”的字体字形，将其轮廓用做几何体。对字符串“2D”的形状进行 30° ($\pi/6$) 的旋转。关于Java 2D编程的细节描述，将在后续章节中进行介绍。

经常将Java程序同时写成一个Java小程序

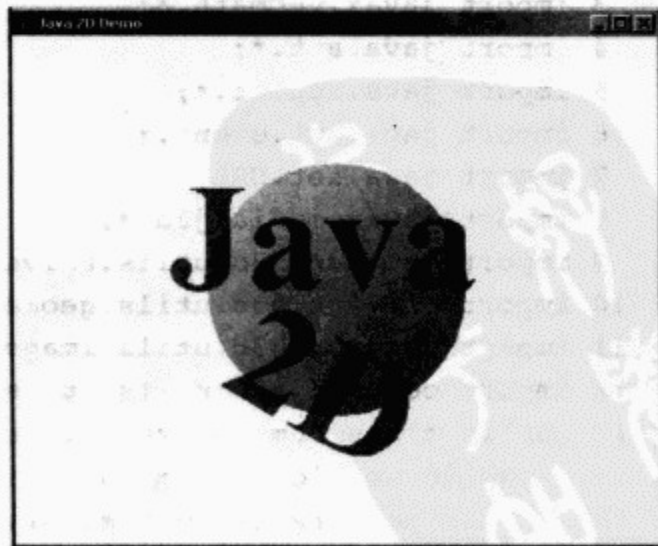


图1-10 Java 2D程序。绘制的圆采用渐变涂色方法进行填充且文字是半透明的

22

(applet) 和一个Java应用程序 (application)。本例就是这样一个具有“双重目的”的程序。Demo2D类是JApplet (第9行) 的一个子类, 它能够作为一个Java小程序来执行。不过, 它还包括了main方法 (第10行), 所以它也可以作为一个Java程序来执行。main 方法生成JFrame的一个实例, 并且把一个Demo2D类的实例加入到窗口中。它通过调用init方法来模仿Java小程序的执行, 它们的结果几乎完全相同。本书中, 大多数例子都将采用这种形式加以展示。

Java 2D是Java 2 平台的一个标准的核心部分。任何Java 2 标准版 (Java 2 Standard Edition, J2SE) 的软件开发包 (Software Development Kit, SDK) 或Java运行时环境 (Runtime Environment, JRE) 的安装, 都自动包含了Java 2D。在这样的SDK环境下, 上面所举的例子不需要额外的包就可以编译。

1.5 Java 3D

Java 3D 是Java平台的可选包, 可以从<http://www.javasoft.com/java3d>上获得。Java 3D提供了令人难以置信的综合性3D图形框架, 包含动画、3D交互和复杂视图等高级功能。同时, 它还提供了相对简单直观的编程接口。

Java 3D程序范型与Java 2D十分不同, 它严格地遵循“建模-绘制”范型。一个被称为场景图 (scene graph) 的抽象模型, 用来组织和维护虚拟场景中的可视对象及其行为。场景图包含了虚拟图形世界的全部信息, Java 3D绘制引擎会对场景图进行自动绘制。

Java 3D在一个Canvas3D对象上进行场景绘制。Canvas3D是一种重量级的组件, 它不能够很好地在新型Swing组件上执行。由于这个原因, 本书中的Java 3D例子都是运用AWT对象来实现的。

注意 Java 3D仍然不使用轻量级画布的原因, 是为了利用硬件加速功能。对于重量级组件, 可以自动地使用由本地平台支持的图形硬件加速功能。虽然可以混合使用重量级和轻量级组件, 但这样做必须注意避免一些不良后果。更多细节可以参考附录B。

程序清单1-9是一个简单的Java 3D应用示例。该程序显示一个旋转的地球, 并在地球前显示一个3D字符串“Java 3D”。程序的运行结果如图1-11所示。

程序清单1-9 Demo3D.java

```
1 package chapter1;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.applet.*;
6 import java.awt.event.*;
7 import java.net.URL;
8 import javax.media.j3d.*;
9 import com.sun.j3d.utils.universe.*;
10 import com.sun.j3d.utils.geometry.*;
11 import com.sun.j3d.utils.image.*;
12 import com.sun.j3d.utils.applet.MainFrame;
13 public class Demo3D extends Applet { //定义Demo3D类, 继承自JApplet类
14     public static void main(String[] args) {
15         new MainFrame(new Demo3D(), 480, 480); //生成主窗口并设置大小
16     }
17
18     private SimpleUniverse su; //声明SimpleUniverse对象变量
```



```
19 //重写初始化函数
20 public void init() {
21     GraphicsConfiguration gc =
22     SimpleUniverse.getPreferredConfiguration();
23     Canvas3D cv = new Canvas3D(gc); //创建一个Canvas3D对象
24     setLayout(new BorderLayout());
25     add(cv);
26     BranchGroup bg = createSceneGraph(); //创建场景图的根节点
27     bg.compile(); //编译场景图
28     su = new SimpleUniverse(cv); //创建和设置SimpleUniverse对象
29     su.getViewingPlatform().setNominalViewingTransform();
30     su.addBranchGraph(bg);
31 }
32
33 public void destroy() {
34     su.cleanup();
35 }
36
37 private BranchGroup createSceneGraph() {
38     BranchGroup root = new BranchGroup(); //创建根节点
39     TransformGroup spin = new TransformGroup(); //创建单位矩阵的变换组
40     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
41     //设置变换组有可写的能力,以便旋转
42     root.addChild(spin); //将其作为根节点的子节点
43     // 3D文本
44     Appearance ap = new Appearance(); //默认表面属性对象
45     ap.setMaterial(new Material()); //设置材质信息,以便进行光照计算
46     Font3D font = new Font3D(new Font("Helvetica", Font.PLAIN, 1),
47     new FontExtrusion()); //设置字体
48     Text3D text = new Text3D(font, "Java 3D");
49     Shape3D shape = new Shape3D(text, ap); //将表面属性对象赋予文本
50     // 进行文本变换
51     Transform3D tr = new Transform3D();
52     tr.setScale(0.2);
53     tr.setTranslation(new Vector3d(-0.35, -0.15, 0.75));
54     TransformGroup tg = new TransformGroup(tr);
55     root.addChild(tg);
56     tg.addChild(shape); //添加对象节点
57     // 球体
58     ap = createAppearance(); //创建表面属性对象,并赋予球体
59     spin.addChild(new Sphere(0.7f,
60     Primitive.GENERATE_TEXTURE_COORDS, 50, ap));
61     // 旋转
62     Alpha alpha = new Alpha(-1, 6000); //创建构造时间变化函数的对象
63     RotationInterpolator rotator =
64     new RotationInterpolator(alpha, spin); //创建旋转对象
65     BoundingSphere bounds = new BoundingSphere();
66     rotator.setSchedulingBounds(bounds); //设定行为活跃区域为范围球体
67     spin.addChild(rotator);
68     //设置背景和光照
69     Background background = new Background(1.0f, 1.0f, 1.0f); //创建背景对象
70     background.setApplicationBounds(bounds);
71     root.addChild(background); //添加背景
72     AmbientLight light =
73     new AmbientLight(true, new Color3f(Color.red));
74     light.setInfluencingBounds(bounds); //设定环境光照范围
```

```

74     root.addChild(light); //添加环境光源节点
75     PointLight ptlight = new PointLight(new Color3f(Color.white),
76     new Point3f(3f,3f,3f), new Point3f(1f,0f,0f));
77     ptlight.setInfluencingBounds(bounds); //设定点光源范围
78     root.addChild(ptlight); //添加点光源节点
79     return root;
80 }
81 //创建外观对象
82 private Appearance createAppearance(){
83     Appearance ap = new Appearance();
84     URL filename =
85     getClass().getClassLoader().getResource("images/earth.jpg");
86     TextureLoader loader = new TextureLoader(filename, this);
87     ImageComponent2D image = loader.getImage(); //打开纹理图片
88     Texture2D texture =
89     new Texture2D(Texture.BASE_LEVEL, Texture.RGBA,
90     image.getWidth(), image.getHeight());
91     texture.setImage(0, image);
92     texture.setEnabled(true);
93     texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
94     texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
95     ap.setTexture(texture); //设定表面纹理
96     return ap;
97 }
98 }

```

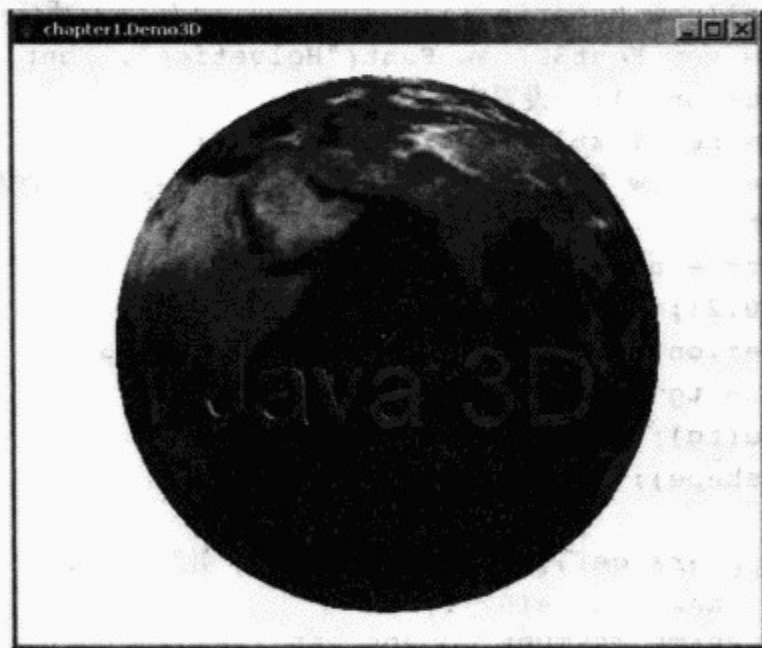


图1-11 一个Java 3D示例，显示一个可旋转的地球和一串3D文本

这是一个典型的Java 3D应用程序。该程序的主要任务集中于构造一个称为场景图的概念数据结构。程序的视觉效果实现方式为，生成一个场景图，并在图中设置恰当的图形元素。程序的场景图如图1-12所示。这是一个树状结构，包括的对象有球体、3D文本、外观、变换、背景和光照等。

场景的绘制由Java 3D引擎自动完成。绘制结果如图1-11所示。关于Java 3D中用场景图编程的相关概念和技术，会在以后的章节进行介绍。

该程序也可以编写成一个双重目的Java小程序/Java应用程序。Demo3D类定义为Applet的子类。main方法也存在于Demo3D中，以便把这个类作为Java应用程序来运行。Java 3D中的一个MainFrame功能类（第16行）提供了在窗口中运行Java小程序的功能。

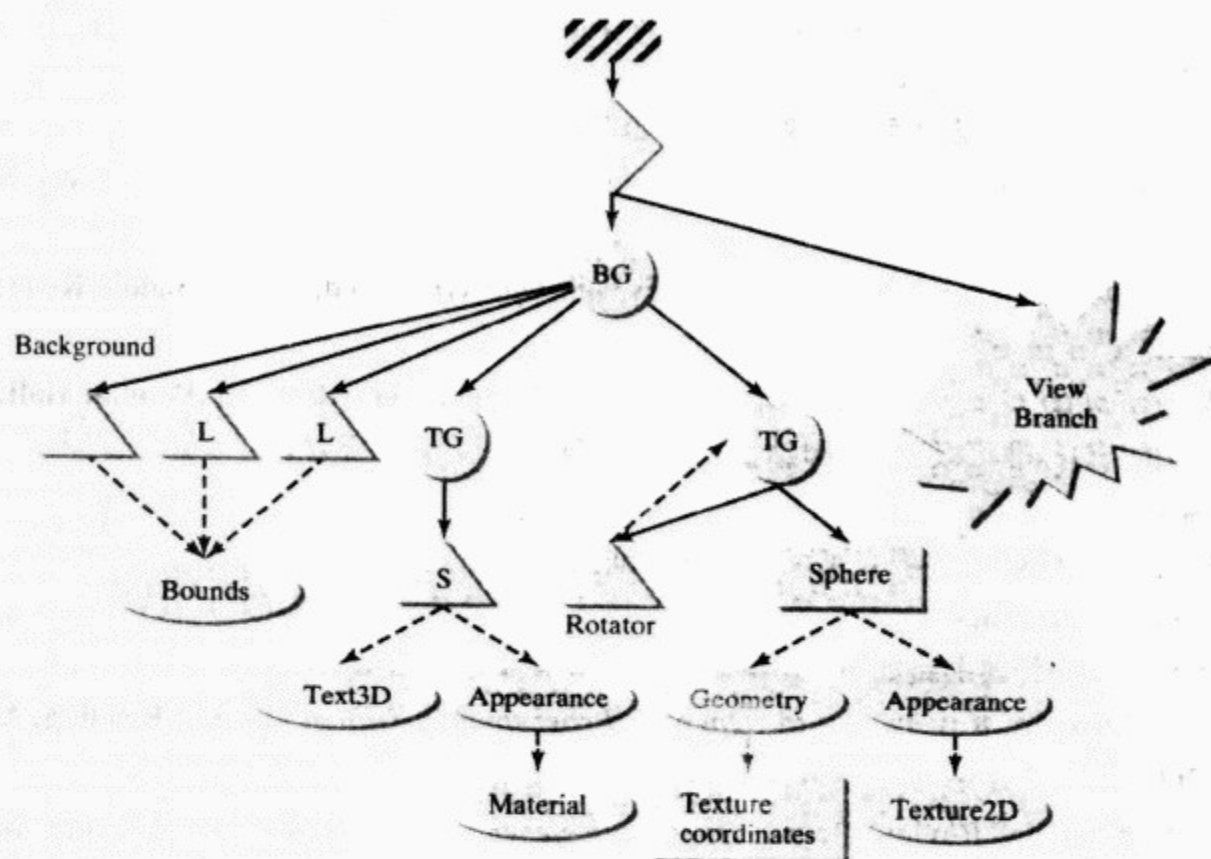


图1-12 Java 3D程序的场景图

1.6 相关领域

计算机图形学、图像处理和计算机视觉都是与计算机相关的领域，它们都处理图形对象。虽然它们的目标和技术是不同的，但是，它们彼此之间存在关联，并且它们之间的分界线变得越来越模糊。

图像处理 (image processing) 是对数字光栅图像进行处理的一种技术。它处理的典型问题包括图像增强、噪音消除、图像压缩和边界检测等。图像处理以一个已有的图像作为输入，对其进行恰当的操作，而计算机图形学则是从虚拟世界中生成一个合成的图像。

图像处理与计算机图形学是紧密相关的。图形绘制的结果通常可以是一幅图像。光栅图像也可以被当做图形基元用到计算机图形中，它们也可以作为纹理来增强图形绘制的真实感。比如，程序清单1-9所绘制的地球，是通过先生成一个球体，然后将地球图像映射到球体表面而得到的。基本的图像处理技术将在第4章进行介绍。

计算机视觉 (computer vision) 试图对真实世界的图像进行理解。在某种程度上说，计算机视觉系统是与计算机图形学系统正好相反的，它的主要目的是从真实世界的图像中重构一个虚拟世界。因此，计算机视觉和计算机图形学彼此是互补的，它们对同一系统提供不同的观察角度。

计算机图形学的理论和实践，对某些重要的数学概念的依赖性很强。与之紧密相关的数学领域包括解析几何和线性代数等。解析几何 (analytic geometry) 为图形对象提供了相关的数学表示形式。线性代数 (linear algebra) 研究对向量空间的操作和变换，对于计算机图形学的许多基本问题都起着重要的作用。相关的数学问题，将在图形学问题的相关章节中进行介绍。附录A对与图形学相关的数学背景知识进行了概要性总结。

1.7 参考资料

一篇与计算机图形学相关的经典参考文献如下：

- J. Foley, A. van Dam, S. Feiner, and J. Hughs, *Computer Graphics, Principles and Practices*, 2d ed., Reading, MA: Addison-Wesley, 1990.

目前, 一些使用OpenGL的计算机图形学教材如下:

- E. Angel, *Interactive Computer Graphics, A Top-Down Approach with OpenGL*, 2d ed., Reading, MA: Addison-Wesley, 2000.
- D. Hearn and M. P. Baker, *Computer Graphics with OpenGL*, 3d ed., Upper Saddle River, NJ: Prentice Hall, 2003.
- F. S. Hill, *Computer Graphics Using OpenGL*, 2d ed., Upper Saddle Rive, NJ: Prentice Hall, 2001.

GKS 和其他的ISO标准可以在ISO网站上找到, 网址为:

[http:// www.iso.ch](http://www.iso.ch)

NCAR 图形包包含GKS的实现, 可以在以下网址找到:

[http:// ngwww.ucar.edu/ng4.4/](http://ngwww.ucar.edu/ng4.4/)

一些关于OpenGL的经典书籍如下:

- OpenGL Architecture Review Board, *OpenGL Programming Guide*, 4th ed., Reading, MA: Addison-Wesley, 2004.
- OpenGL Architecture Review Board, *OpenGL Reference Manual*, 4th ed., Reading, MA: Addison-Wesley, 2004.

有许多与OpenGL有关的网站。OpenGL官方网站包含了一些有用的信息和链接, 网址为:

<http://opengl.org>

有关JOGL项目的相关网站网址为:

<http://jogl.dev.java.net>

可以在Java开发商的官方网站上找到相应的教程、文档、软件下载和其他有用信息, 网址为:

<http://java.sun.com>

下面是一本有用的Java 3D参考书, 它给出了API的规范:

- H. Sowizral, K. Rushforth, and M. Deering, *The Java 3D API Specification*, 2d ed., Reading MA: Addison-Wesley, 2000.

下面是一些有关Java 3D 的网络资源:

<http://java.sun.com/products/java-media/3D/index.jsp>

<http://j3d.org>

<https://java3d.dev.java.net/>

主要的类和方法

- **javax.swing.JFrame** Swing中的一个类, 用于应用程序主窗口。
- **javax.swing.JFrame.setDefaultCloseOperation(int)** 该方法设置对窗口关闭事件的响应。
- **java.lang.System.exit(int)** 该方法用于终止程序。
- **java.awt.Frame** AWT中的一个类, 用于应用程序主窗口。
- **javax.swing.JPanel** 一个Swing组件, 可以作为容器或者作为自定义画布的基类。
- **javax.awt.Graphics** 用于AWT中所有图形绘制工具的类。
- **javax.awt.event.MouseListener** 该接口用来监听和处理鼠标事件。
- **javax.awt.event.MouseListener.mouseClicked(MouseEvent)** 该方法实现对鼠标点击事件的处理。

关键术语

- **建模 (modeling)** 构造图形模型的过程。
- **绘制 (rendering)** 从图形模型构造一幅场景图像的过程。

- **虚拟世界** (virtual world) 在计算机内构造的图形模型。
- **API** 应用程序接口。一个标准化的软件接口，定义如何使用一个软件包的功能。
- **GKS** 图形核心系统。一种标准图形API。
- **PHIGS** 程序员的分层交互式图形系统。一种标准图形API。
- **OpenGL** 一种标准图形API，从Silicon Graphics公司的GL图形库演变而来，它有一个通常与C语言关联的编程接口。
- **JOGL** OpenGL的一种Java语言绑定。
- **图像处理** (image processing) 电子工程与计算机科学中的一个领域，研究数字图像的计算机处理。
- **计算机视觉** (computer vision) 计算机科学与工程中的一个领域，研究从获取的图像中感知和重建场景。
- **AWT** 抽象窗口工具包。从早期版本的Java API开始就存在的Java图形包。
- **Swing** 一个新的增强的Java 图形包。
- **OOP** 面向对象编程。一种软件工程范型，它将程序看做是一个由相互关联的对象所组成的系统。

本章提要

- 本章对计算机图形学进行了概述，并对它的基本结构和应用，以及它与其他相关领域的关系进行了介绍。此外，还对Java平台的图形支持的相关内容进行了概述。
- 计算机图形学的基本目标，包括构建图形对象的虚拟世界和对虚拟世界的场景进行绘制。建模和绘制是计算机图形学的两大主题。
- 2D和3D图形学系统所包含的问题特性是完全不同的。通常，2D和3D图形学系统有着不同的结构，它们可以通过单独的包来实现。因为这个原因，在Java平台上，Java 2D和Java 3D是两个独立的包，它们有着不同的编程模型。
- 图形学编程环境由面向硬件的低层方法，发展为面向对象的高级范型。为了减少对平台的依赖性，并得到更高层次的抽象，人们相继开发了大量的图形API。典型的标准化图形包有GKS、PHIGS、OpenGL、Java 2D和Java 3D。
- 与早期的Java版本不同，Java 2 平台对图形提供广泛的支持。Java 2D和Java 3D包是功能完备的高级图形学系统。本书将以它们作为主要工具，对图形编程进行介绍。
- 计算机图形学作为与计算机相关的一门学科，不同于图像处理和计算机视觉。不过，它们之间是相关联的，都要对可视图像进行处理。

28

复习题

- 1.1 列举出三种用到2D计算机图形学的应用。
- 1.2 列举一种用到3D计算机图形学的非游戏类应用。
- 1.3 在网上搜索运用计算机图形学的电影。
- 1.4 识别下面列举的应用属于哪个领域（计算图形学、图像处理和计算机视觉）：
 - (a) 在乳房X线照片上定位小亮斑。
 - (b) 从一组照片中构造建筑物的3D模型。
 - (c) 模拟显示太阳系，包括太阳和运动中的九大行星。
 - (d) 在MRI扫描图中识别脑部区域，并显示脑部的3D模型。
 - (e) 用计算机生成汽车碰撞场景。
 - (f) 运用计算机从照片中识别人的身份。
- 1.5 在网上搜索运用GKS编程的实例。
- 1.6 在网上搜索运用PHIGS编程的实例。
- 1.7 比较OpenGL和Java 3D，列举出它们各自的优点。

1.8 讨论在如Java之类的标准语言平台中包含图形支持的优缺点。

1.9 列举出AWT的主要GUI组件，并在Swing中寻找与其对应的组件。

29 1.10 阅读Java 3D的文档，列举出Java 3D中的Java 包。

编程练习

1.1 写一个基于控制台的Java程序，用100个随机数填充一个double类型数组，并打印出它们的平均值及标准差。

1.2 写一个Java AWT程序，在窗口中央绘制一个圆。

1.3 写一个Java Swing程序，在窗口中央绘制一个圆。

1.4 写一个Java GUI程序，对鼠标点击进行响应，在鼠标点击的位置画一个填充的圆。

1.5 在自己的机器上编辑、编译、运行程序清单1-8所显示的Java 2D程序。

30 1.6 在自己的机器上编辑、编译、运行程序清单1-9所显示的Java 3D程序。

第2章 2D图形学：基础

学习目标

- 了解2D图形系统的体系结构及其操作。
- 理解图的2D坐标系及其方程。
- 能辨别绘制流水线中的各种坐标空间。
- 熟悉Java 2D程序结构及Graphics2D对象。
- 能运用Java程序实现图方程。
- 能运用基本的2D几何基元。
- 能使用GeneralPath类构建自定义几何体。
- 能通过构造区域几何模型来构建几何体。

31

2.1 引言

本章将介绍与2D图形系统相关的基本概念和Java 2D包中所包含的内容。2D图形系统在2D空间中对虚拟世界进行建模。与3D图形相比，2D图形在模型和绘制方面都更简单，2D对象的生成和操作更容易，而且，2D图形的绘制过程不需要像3D图形那样，通过使用复杂的投影来实现。虽然2D模型不能完全表现3D空间的全部特性，但由于2D计算机图形具有简单与高效的特性，所以得到广泛应用。2D计算机图形是现代基于GUI程序的一个重要组成部分。

与2D图形相关的基本概念包括：绘制流水线、对象空间、世界空间、设备空间、坐标系、图元、几何变换、颜色、裁剪、合成规则等。Java 2D为2D图形提供了全面的支持。本章对Java 2D程序的基本结构和几何对象模型进行了介绍。关于2D图形和Java 2D编程的其他相关内容，将在接下来的两章中进行详细的讨论。

2.2 2D图形绘制过程

在2D图形学中，虚拟世界空间和视图空间都是2D的。绘制（rendering）是指通过一些相对简单的变换来形成各种对象。通常，甚至不需要用一个2D世界空间来明确地表示图形对象之间的相互关系。但是，为了得到清晰的系统结构并与3D图形保持一致，仍然保留了虚拟世界空间的概念。

概念上，一个图形对象可以在它自己的对象空间（object space）中进行定义，然后通过对象变换放置到2D的世界空间（world space）中。2D图形的绘制任务是对虚拟世界进行快照，并在设备空间中生成一幅表示特定视图的图像。图2-1给出了2D图形典型的绘制过程。

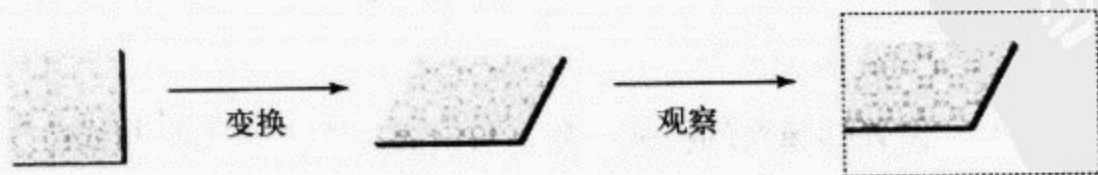


图2-1 2D图形对象在变换和观察流水线上的处理情形

2D图形系统的主要组成部分，包括所要绘制的2D图形对象模型、应用于图形对象的几何

变换,以及在显示设备上绘制虚拟世界特定视图的绘制引擎。简单的2D图形程序进行图形绘制的基本步骤如下:

- 1) 构建2D图形对象。
- 2) 对构建的图形对象进行几何变换。
- 3) 应用颜色和其他绘制属性。
- 4) 在图形设备上绘制场景。

模型中的图形对象是2D的。除了由直线、多边形、椭圆等基本图元所构建的几何对象之外,模型中还可以包含文本和图像等对象。

2D图形所涉及的变换,通常是仿射变换。对象变换可以改变虚拟对象的形状和位置。观察变换 (viewing transformation) 虽然并不改变虚拟世界模型,但是能够改变世界空间中整个场景的视图。例如,对于含有一个圆和一个三角形的虚拟世界模型来说,对圆施加一个平移变换作为对象变换,这将仅改变圆的位置,而对三角形没有任何影响。但是,如果这个平移变换是作为一个观察变换,则移动的将是整个视图。

32

除了几何属性外,很多其他属性对场景的绘制也有一定的影响。这类属性的典型例子是颜色、透明度、纹理与线型等。在2D图形系统中,对于场景的绘制是基于几何信息、变换以及由其他属性构成的上下文来完成的。

2.3 2D几何模型与坐标系

图形模型的基本组件是几何对象。为了在计算机上精确而高效地表示几何信息,就需要用到各种坐标系 (coordinate system)。最常用的2D坐标系是直角 (笛卡儿) 坐标系,如图2-2所示。

2D坐标系在平面上有两个相互垂直的坐标轴,每个坐标轴都以一组实数标注。通常,水平轴称为x轴,垂直轴称为y轴。两条轴的交点在两条坐标轴上都标为0,称为原点。平面上的每个点都和一对实数 (x, y) 相关联,称为该点的x坐标和y坐标。坐标度量了该点相对于坐标轴的水平 and 垂直位置。

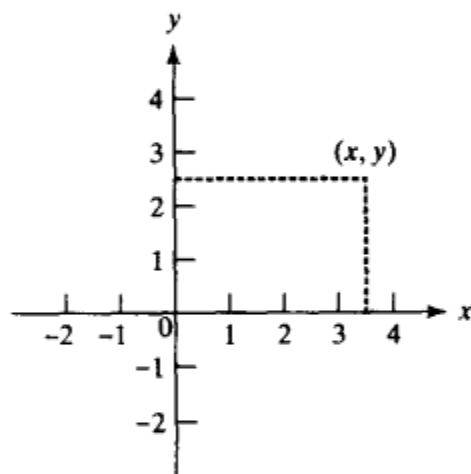


图2-2 一个2D坐标系

2D几何对象是平面上点的集合,在组成几何对象的集合中,点的数量通常是无限的。为了有效地表示这样的对象,可以通过方程来定义构成对象的点的x坐标与y坐标之间,必须满足的关系。

例如,一条直线 (line) (如图2-3所示) 可以通过一次多项式方程来表示 (线性方程):

$$Ax + By + C = 0$$

33

一个圆心在原点、半径为 R 的圆,可以通过以下方程进行表示:

$$x^2 + y^2 = R^2$$

一个中心在 (x_0, y_0) 位置的椭圆的标准方程,可以表示为:

$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1$$

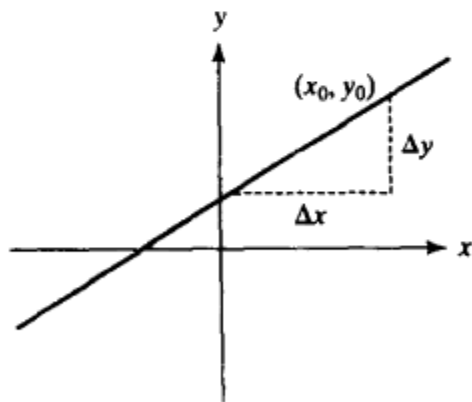


图2-3 用线性方程表示的直线

另外一种常用的曲线方程是参数方程 (parametric equation)，它用第三个变量 t 来代替方程中的 x 变量和 y 变量。 x 、 y 可以用关于 t 的函数来表示，其表示形式为：

$$x = f(t)$$

$$y = g(t)$$

参数方程的优点之一是，它们能够用显函数的形式来估计坐标。如图2-4所示的椭圆，也可以采用参数方程的形式进行表示，其表示形式为：

$$x = x_0 + a \cos t$$

$$y = y_0 + b \sin t$$

由所有点或坐标构成的集合称为空间 (space)。图形系统一般涉及三个类型的空间，它们是对象空间 (object space)、世界空间 (world space) 和设备空间 (device space)。每个空间通常由其自身的坐标系来表征。在一个空间中的几何对象，可以通过几何变换映射到另一个空间中。

对象坐标系也可以称为本地坐标系 (local coordinate system) 或建模坐标系 (modeling coordinate system)，它和某个特定的图形对象或图形基元相关联。在构造这样一个对象时，通常不去考虑它在世界空间中的最终位置和外观，而选择一个对该对象更自然的坐标系，这样会更为方便。例如，定义一个圆基元时，可以选择坐标系的原点作为圆心，并简单地定义一个单位圆 (半径为1的圆)。这个圆以后可以通过称为平移 (translation) 变换的几何变换，移动到世界空间中的任何位置。它的半径可以通过缩放而变成任意值，甚至可以通过非均匀缩放来将其变成一个椭圆。

世界坐标系 (world coordinate system) 又称为用户坐标系 (user coordinate system)，它为模型中所有的图形对象定义了一个公共参考空间。世界坐标系定义了由建模和绘制子系统共享的一个虚拟世界，通过对象变换可以将几何对象移到该空间中。绘制系统获取该空间的快照，并在输出设备上生成绘制图像。

设备坐标系 (device coordinate system) 表示显示屏幕或打印机等输出设备的显示空间。图2-5给出了这种坐标系的典型例子。原点位于左上角， y 轴的正方向是向下的，坐标值通常只能是整数。这种情况明显不同于通常的数学表示形式，不过，对于多数计算显示设备来说，这样的表示方法更为自然。

默认情况下，Java 2D的世界坐标与设备坐标是一致的。运用图形系统中的几何变换能力，很容易定义一个不同的世界空间，以满足某个特定的应用需要。

2.4 Graphics2D 类

在Java 2D中，可以通过Graphics2D类来访问绘制引擎。在早期的Java版本中，图形的绘制是通过java.awt.Graphics类来完成的。Graphics类中包含了绘制图形基元和控制绘制模式的基本方法。Java 2D使用功能更全面的Graphics2D类来绘制图形，它继承了Graphics类，以兼容早期

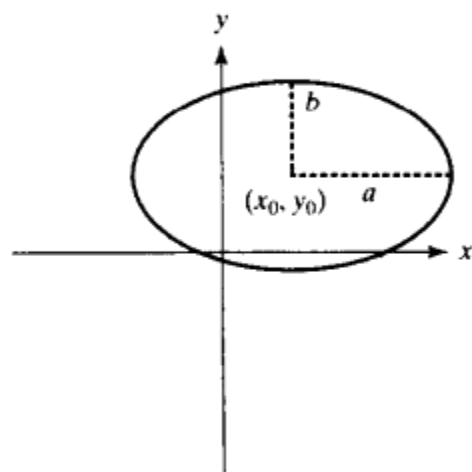


图2-4 用二次方程表示的椭圆

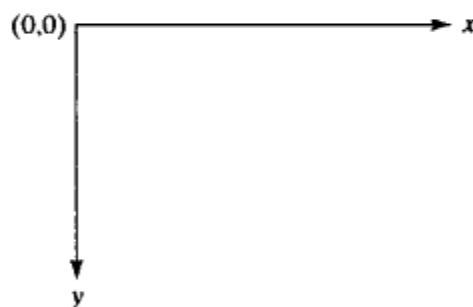


图2-5 Java 2D的坐标系，从原点 (0, 0) 出发， x 轴向右增长， y 轴向下增长

AWT包中GUI组件的绘制功能。

Graphics类和Graphics2D类都是抽象类，原因在于它们的实现很有必要是平台相关的。因此，不能直接实例化Graphics2D类。有两种方法可以得到Graphics2D类对象，一种方法是从paintComponent方法的参数获得，另外一种方法是通过调用getGraphics方法来获得。在JComponent对象上绘制图形的标准方法是重写paintComponent方法：

```
void paintComponent(Graphics g)
```

参数g声明为Graphics对象，同时，它也是一个Graphics2D对象。它能够被强制转换为Graphics2D对象。只要显示的内容需要进行重绘，例如当最小化的窗口恢复原状时，Java虚拟机就会自动调用paintComponent方法。因此，这个方法所实现的绘制效果看起来是持久性的。

Java并没有为2D图形提供“保留模式”的建模功能。paintComponent方法中的自定义代码，可以作为图形系统的隐式模型。

另外一种获得Graphics2D对象的方法是，通过调用Component类中的getGraphics方法来实现。

35

```
Graphics getGraphics()
```

同样，从此方法中返回的Graphics对象可以强制转化为Graphics2D的对象，它能够用来进行图形的绘制。但是，这种方法得到的绘制结果通常不是持久的。

在AWT中，Graphics类中提供的方法可以控制绘制的各个方面。所包括的方法可以设置颜色和字体、平移坐标、设置XOR模式，以及直接画线条和椭圆等图元。Graphics类的一些常用方法如下：

```
void setColor (Color c) //将图形上下文的当前颜色设置为指定颜色。使用此图形上下文的所有后续图形操作，均使用这个指定的颜色。
void setFont (Font f) //将图形上下文的字体设置为指定字体。使用此图形上下文的所有后续文本操作，均使用此字体。
void setXORMode (Color c) //将此图形上下文的绘图模式，设置为在此图形上下文的当前颜色和新的指定颜色之间交替显示。这指定了逻辑像素操作以 XOR 模式执行，在此模式中，像素在当前颜色和指定的 XOR 颜色之间交替显示。
void setPaintMode () //设置此图形上下文的绘图模式，以便通过此图形上下文中的当前颜色来覆盖目标。此方法将逻辑像素操作函数设置为绘图模式或覆盖模式，所有后续呈现操作将使用当前颜色覆盖目标。
void translate (int x, int y) //将Graphics2D上下文的原点平移到当前坐标系中的点(x, y)。
void drawLine (int x1, int y1, int x2, int y2) //在此图形上下文的坐标系中，使用当前颜色在点(x1,y1)和(x2,y2)之间画一条线。
void drawRect (int x1, int y1, int width, int height) //绘制指定矩形的边框。矩形的左边缘和右边缘分别位于x和x+width。上边缘和下边缘分别位于y和y+height。使用图形上下文的当前颜色绘制该矩形。
void drawOval (int x1, int y1, int width, int height) //绘制椭圆的边框。得到一个圆或椭圆，它刚好能放入由x、y、width和height参数指定的矩形中。椭圆覆盖区域的宽度为 width+1像素，高度为 height+1像素。
void drawArc (int x1, int y1, int width, int height, int start, int arc) //绘制一个覆盖指定矩形的圆弧或椭圆弧边框。得到的弧从start开始跨越arc度，并使用当前颜色。弧的中心是矩形的中心，此矩形的原点为 (x,y)，大小由width和height参数指定。得到的弧覆盖 width+1 像素宽乘以 height+1 像素高的区域。
void drawRoundRect (int x1, int y1, int width, int height, int arcW, int arcH) //用此图形上下文的当前颜色绘制圆角矩形的边框。矩形的左边缘和右边缘分别位于x和x+width，矩形的上边缘和下边缘分别位于y和y+height。
void drawPolygon (int [] xPoints, int[] yPoints, int nPoints) //绘制指定的多边形边框。
void fillRect (int x1, int y1, int width, int height) //填充指定的矩形。该矩形左边缘和
```

右边缘分别位于x和x+width-1。上边缘和下边缘分别位于y和y+height-1。得到的矩形覆盖width像素宽乘以height像素高的区域。使用图形上下文的当前颜色填充该矩形。

```
void fillOval (int x1, int y1, int width, int height) //使用当前颜色填充外接指定矩形框的椭圆。
void fillArc (int x1, int y1, int width, int height, int start, int arc) //填充覆盖指定矩形的圆弧或椭圆弧。
void fillRoundRect (int x1, int y1, int width, int height, int arcW, int arcH) //用当前颜色填充指定的圆角矩形。矩形的左边缘和右边缘分别位于x和x+width-1。矩形的上边缘和下边缘分别位于y和y+height-1。
void fillPolygon (int [] xPoints, int[] yPoints, int nPoints) //填充由x和y坐标数组定义的闭合多边形。
void drawString(String str, int x, int y) //使用Graphics2D上下文中的当前文本属性状态呈现指定的String的文本。
```

在AWT中，还没有明确地区分建模与绘制。例如，一个简单的方法drawOval，同时负责定义（建模）一个椭圆和绘制这个椭圆。

与此相对照，Java 2D需要处理复杂得多的图形对象，因此，并没有将所有的绘制函数封装到一个类中，而是将建模和几何变换的功能由独立于Graphics2D类的其他类来实现。Graphics2D提供了一些通用的方法，例如draw (Shape) 和fill (Shape)，用于绘制单独定义的各种图形对象。需要绘制的各种图形对象实现为Shape对象。类似地，各种几何变换可以用AffineTransform类来构造。Graphics2D通过setTransform (AffineTransform) 方法，把当前的几何变换设置为某个单独定义的几何变换对象。Graphics2D类提供的部分方法如下：

```
void draw (Shape s) //使用当前 Graphics2D 上下文的设置，勾画 Shape 的轮廓
void fill (Shape s) //使用 Graphics2D 上下文的设置，填充 Shape 的内部区域
void setTransform (AffineTransform Tx) //重载Graphics2D 上下文中的变换
void transform (AffineTransform Tx) //根据"最后指定首先应用"规则，使用此 Graphics2D 中的变换组合 AffineTransform 对象。
void setPaint(Paint p) //为 Graphics2D 上下文设置 Paint 属性。
void setStroke (Stroke s) //为 Graphics2D 上下文设置 Stroke。
void clip (Shape s) //将当前 Clip 与指定 Shape 的内部区域求交集，并将 Clip 设置为所得的交集。
void setComposite (Composite c) //为Graphics2D 上下文设置 Composite。
void addRenderingHints (Map hints) //为渲染算法设置任意数量的首选项值。
```

在Graphics2D中，对建模与绘制的分离是十分明显的。例如，如果要绘制一个椭圆，则首先需要创建Ellipse2D类的一个实例（实现了Shape接口），然后调用Graphics2D对象的draw方法来绘制它。

36

典型的Java 2D图形程序，用JPanel类作为绘制画布。通过重载paintComponent方法，可以实现自定义绘制。相关联的Graphics2D对象，可以对颜色、画笔、笔画和变换等进行恰当的设置。图形对象作为一个实现了Shape接口的类的实例对象加以构造，并通过Graphics2D类对象来绘制。

程序清单2-1给出了一个简单的Java 2D程序。它演示了Java 2D程序的基本结构以及Graphics2D类的使用情形。程序的执行结果如图2-6所示，它所显示的内容为一个进行了几何变换的圆和一个蓝色字符串。

程序清单2-1 Hello2D.java

```
1 package chapter2;
2
3 import java.awt.*;
```



```
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.geom.*;
7 //定义Hello2D类, 继承自JApplet类
8 public class Hello2D extends JApplet {
9     public static void main(String s[]) {
10         JFrame frame = new JFrame();//创建主窗口
11         frame.setTitle("Hello 2D");//设置标题栏
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         JApplet applet = new Hello2D();//生成Hello2D实例并初始化
14         applet.init();
15         frame.getContentPane().add(applet);//将Hello2D实例加入frame内容窗格
16         frame.pack();
17         frame.setVisible(true);//设置窗口可见
18     }
19     //重写初始化函数
20     public void init() {
21         JPanel panel = new Hello2DPanel();
22         getContentPane().add(panel);
23     }
24 }
25 //定义Hello2DPanel类, 继承自JPanel类
26 class Hello2DPanel extends JPanel {
27     public Hello2DPanel() {
28         setPreferredSize(new Dimension(640, 480));//设置窗口首选大小
29     }
30     //重写组件绘制函数
31     public void paintComponent(Graphics g) {
32         super.paintComponent(g);
33         Graphics2D g2 = (Graphics2D)g; //强制转换为Graphics2D对象
34         g2.setColor(Color.blue); //设置颜色
35         Ellipse2D e = new Ellipse2D.Double(-100, -50, 200, 100);
36         AffineTransform tr = new AffineTransform();
37         tr.rotate(Math.PI / 6.0); //设置旋转变换
38         Shape shape = tr.createTransformedShape(e);
39         g2.translate(300,200); //进行平移变换
40         g2.scale(2,2); //进行缩放变换
41         g2.draw(shape); //绘制可旋转的椭圆
42         g2.drawString("Hello 2D", 0, 0); //绘制字符串
43     }
44 }
```

37

这个程序的写法可以同时作为applet和application运行。Hello2D类继承了JApplet类, 它包含了main方法, 创建了一个JFrame窗口, 并向窗口中加入了一个Hello2D类的实例。这个Applet包含了Hello2DPanel类的一个实例对象。Hello2DPanel类扩展了Jpanel类, 重写了paintComponent方法, 以便实现对图形的绘制。

主要的图形功能在paintComponent方法(第31行)内实现。它首先调用超类的paintComponent方法进行必要的清除操作。然后, 将Graphics类对象g强制转换为Graphics2D类对象g2(第33行), 以便使用Java 2D的特性。参数g虽然声明为Graphics类对象, 但实际上在包括Java 2D在内的所有Java版本里, 它都是Graphics2D类的对象。绘制的图形颜色被设置为蓝色, 这是通过Graphics类的setColor()方法来实现的。关于Graphics2D类中更多复杂的绘制特性, 将在本章后面进行详细介绍。

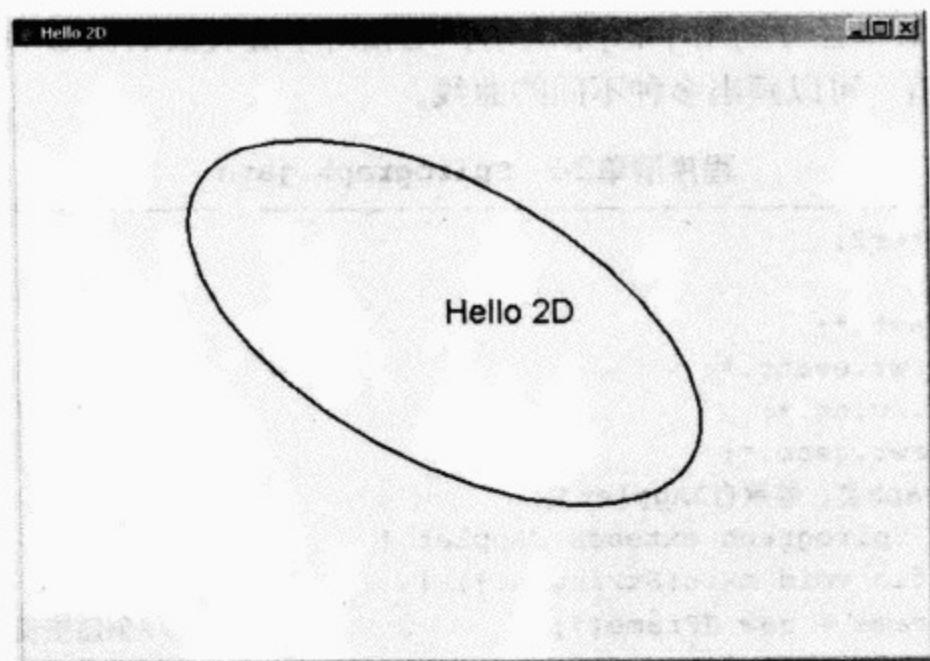


图2-6 一个简单的Java 2D应用程序，绘制一个旋转了一定角度的椭圆和一段文字

用Ellipse2D.Double类（第35行）创建了一个椭圆对象。一个 $\pi/6$ 角度的旋转变换，是作为AffineTransform类对象加以创建的，该旋转只应用到椭圆上。变换后的形状通过调用AffineTransform类对象的createTransformedShape方法来获得。

对象变换包括一个平移变换（300，200）和一个比例因子为2的缩放变换。这两个变换是通过直接调用Graphics2D类的translate和scale方法来实现的。

运用draw方法在屏幕上绘制旋转过的椭圆（第41行）。字符串“Hello 2D”通过调用Graphics2D类的drawString方法来绘制。

2.5 绘图方程

数学方程对于图形对象的建模是十分重要的。相反，作图是研究数学函数和方程的一种有用工具。根据方程作图是一种简单的图形学应用。

给出一个图形方程，为其作图的一个简单方法是，求解一系列满足方程的坐标，然后，绘制这些点。对于一个形式为 $y = f(x)$ 的函数，最为直接的方法是选择一组 x 坐标，然后计算对应的 y 坐标。对于一个隐函数方程 $F(x, y) = 0$ ，它的计算就比较困难了。这是因为，给定一个 x （或 y ）的坐标值，通常需要通过解一个方程，才能计算出另一个坐标值。某些方程可以表示为便于计算的参数形式。

该程序的运行结果如图2-7所示。

程序清单2-2 基于参数方程绘制了一个螺旋线。考虑一个圆在另一个圆上滚动，把一支笔附着于滚动的圆上，所画出的曲线称为外摆线或螺旋线。将滚动的角度作为参数，其对应的参数方程可以表示为：

$$\begin{aligned}x &= (r_1 + r_2) \cos t - p \cos((r_1 + r_2)t/r_2) \\y &= (r_1 + r_2) \sin t - p \sin((r_1 + r_2)t/r_2)\end{aligned}$$

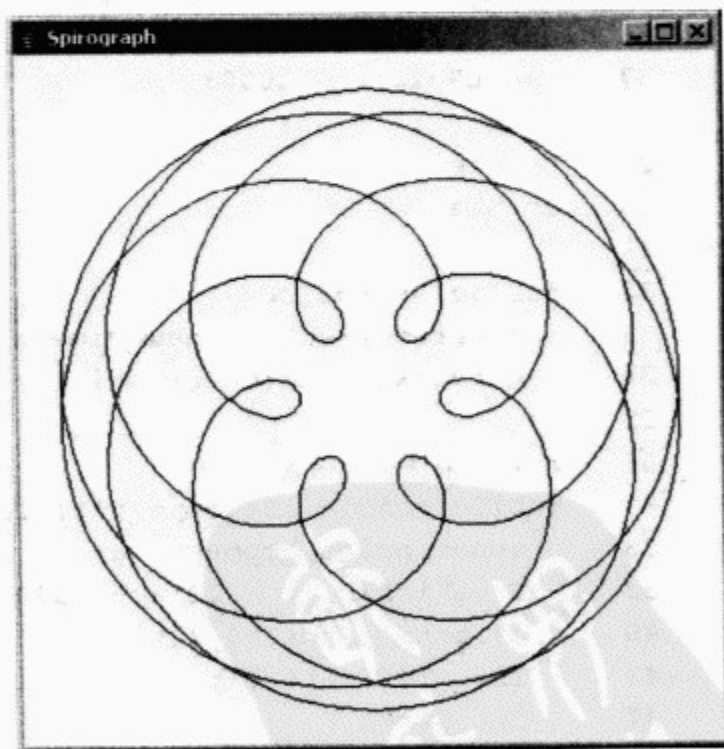


图2-7 通过参数方程绘制的螺旋线

固定圆和滚动圆的半径分别用 r_1 和 r_2 来表示, 笔相对于旋转圆的圆心的偏移量定义为 p 。定义不同的 r_1 , r_2 和 p 的值, 可以画出多种不同的曲线。

程序清单2-2 Spirograph.java

```

1 package chapter2;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.geom.*;
7 //定义Spirograph类, 继承自JApplet类
8 public class Spirograph extends JApplet {
9     public static void main(String s[]) {
10         JFrame frame = new JFrame();           //创建主窗口
11         frame.setTitle("Spirograph");           //设置标题栏
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         JApplet applet = new Spirograph(); //生成Spirograph实例并初始化
14         applet.init();
15         frame.getContentPane().add(applet); //将Spirograph实例加入frame内容窗格
16         frame.pack();
17         frame.setVisible(true);                 //设置窗口可见
18     }
19 //重写初始化函数
20 public void init() {
21     JPanel panel = new SpiroPanel();
22     getContentPane().add(panel);
23 }
24 }
25 //定义SpiroPanel类, 继承自JPanel类
26 class SpiroPanel extends JPanel{
27     int nPoints = 1000;                       //声明类中变量并设定默认值
28     double r1 = 60;
29     double r2 = 50;
30     double p = 70;
31
32     public SpiroPanel() {
33         setPreferredSize(new Dimension(400, 400)); //设置窗口首选大小
34         setBackground(Color.white);               //设置背景颜色为白色
35     }
36 //重写组件绘制函数
37 public void paintComponent(Graphics g) {
38     super.paintComponent(g);
39     Graphics2D g2 = (Graphics2D)g;               //强制转换为Graphics2D对象
40     g2.translate(200,200);                         //进行平移变换
41     int x1=(int)(r1+r2-p);                         //第一个点的坐标值
42     int y1=0;
43     int x2;
44     int y2;
45     for (int i=0; i<nPoints; i++) { //根据参数方程计算各点坐标并连接相邻点
46         double t = i*Math.PI/90;
47         x2 = (int)((r1+r2)*Math.cos(t)-p*Math.cos((r1+r2)*t/r2));
48         y2 = (int)((r1+r2)*Math.sin(t)-p*Math.sin((r1+r2)*t/r2));
49         g2.drawLine(x1, y1, x2, y2);
50         x1 = x2;

```

39

```

51      y1 = y2;
52    }
53  }
54 }

```

SpiroPanel类被定义为JPanel类的子类，作图是在paintComponent方法（第37行）内实现的，变量nPoints定义要计算的点的数量。参数方程的变量定义如下：

```

r1=60
r2=50
p=70

```

参数 t 初始值为0，每过一个点增加 $\pi/9$ 。 x 坐标值和 y 坐标值通过参数方程来计算，相邻点之间通过线段来连接。

2.6 几何模型

组成虚拟世界的2D图形对象，可以包括几何体、文本对象和图像等。关于字体、文本和图像的相关讨论，将在第3章和第4章中进行。在这一节里，将重点讨论几何对象。

2.6.1 形状

在Java 2D中，如果实现了Shape接口，一个几何对象可以由Graphics2D类绘制。Graphics2D类包含draw (Shape s)方法和fill (Shape s)方法，可以用来绘制形状的轮廓，或者对形状的内部进行填充。Java 2D提供的工具可以用来构建基本的形状，并且可以通过形状组合，构建更为复杂的形状。Shape类的层次结构如图2-8所示。

40

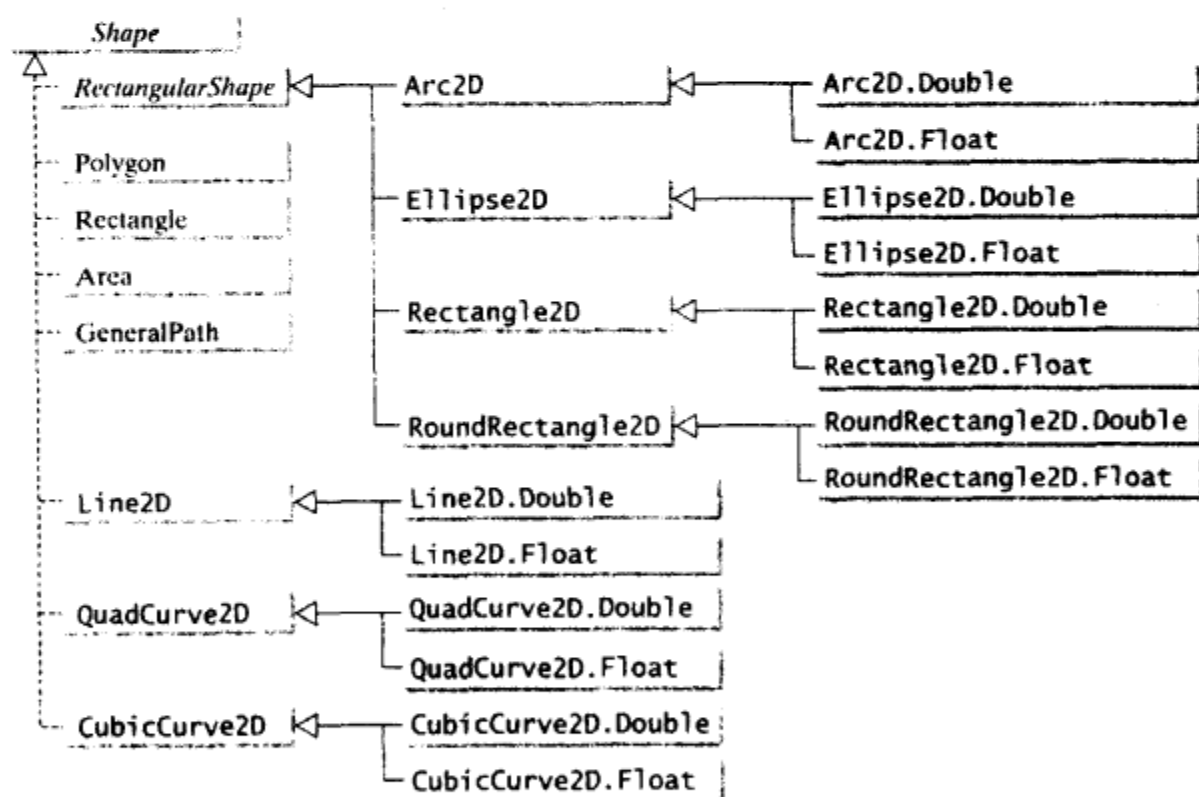


图2-8 Java 2D定义的各种形状

Line2D、QuadCurve2D、CubicCurve2D、Rectangle2D、RoundRectangle2D、Arc2D和Ellipse2D等类，都是抽象类。每个这样的类，都有两个非抽象的内部子类（inner subclasses），它们是X.Double和X.Float。这两个子类，分别用double或float数据类型来表示坐标。例如，Line2D.Double和Line2D.Float是Line2D的子类，同时，它们也是Line2D的内部类。这两个内部子类都表示直线，但是表示坐标的数据类型却是不同的。要生成double数据类型的Line2D对象，

可以用下面的构造方法来完成：

```
Line2D line=new Line2D.Double(x1, y1, x2, y2);
```

QuadCurve2D类表示二次曲线 (quadratic curve)，它在数学上定义为一个二次多项式，通过三个控点来定义。第一个和最后一个控点是曲线的端点。中间的控点通常不在曲线上，但是它却定义了二次曲线的走向，如图2-9所示。QuadCurve2D的对象可以通过下面的构造函数创建，其中三个控点的坐标用六个参数来指定。

41 QuadCurve2D quad=new QuadCurve2D.Double(x1, y1, x2, y2, x3, y3);

CubicCurve2D类表示三次贝塞尔曲线 (cubic Bezier curve)，它由四个控点定义为一个三次多项式。类似于二次曲线，第一个和最后一个控点是曲线的端点。中间的两个控点定义曲线的形状，但不一定在曲线上，如图2-10所示。CubicCurve2D对象可以通过下面的方法进行构造。

```
CubicCurve2D cubic=new CubicCurve2D.Float(x1,y1,x2,y2,x3,y3,x4,y4);
```



图2-9 由三个控点定义的二次曲线

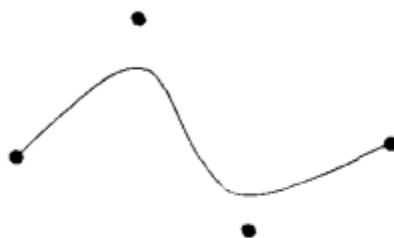


图2-10 通过四个控点定义的三次贝塞尔曲线

Rectangle2D类定义了有水平边和垂直边的矩形 (rectangle)。在JDK 1.1 中定义的Rectangle类已经集成到Java 2D之中，现在是Rectangle2D的子类。Rectangle的坐标都为整数，Rectangle2D.Double 和Rectangle2D.Float是Rectangle2D的另外两个子类，分别用double 类型和float类型数据来表示坐标。下面的代码段，通过不同的数据类型生成三个Rectangle2D对象，矩形的左上角坐标为 (20, 30)，大小为 (100, 80)。

```
Rectangle2D ri=new Rectangle(20,30,100,80)
Rectangle2D rd=new Rectangle2D.Double(20.0,30.0,100.0,80.0);
Rectangle2D rf=new Rectangle2D.Float(20f,30f,100f,80f);
```

RoundRectangle2D类用于定义圆角 (round corner) 矩形。RoundRectangle2D的构造函数有两个额外的参数，用来表示圆角弧的宽和高。例如，下面定义的这个圆角矩形的弧的大小为5×5：

```
RoundRectangle2D rrect=new RoundRectangle2D.Double (20,30,100,80,5,5);
```

Ellipse2D表示一个完整的椭圆 (ellipse)。中心在原点的椭圆，其参数方程可以定义为：

$$\begin{aligned}x &= a \cos \theta \\y &= b \sin \theta\end{aligned}$$

参数 θ 的变化范围从0到 2π ，或从0°到360°。Ellipse2D对象的位置和大小，由它的外接矩形来定义。数据类型为float类型的Ellipse2D对象，可以用下面的构造函数来定义，其外接矩形的左上角坐标为 (20, 30)，大小为100×80。

42 Ellipse2D ellipse=new Ellipse2D.Float(20f,30f,100f,80f);

Arc2D 类用来定义椭圆弧 (arc)。它所在的椭圆定义与Ellipse2D相同，可以用相同的参数方程来表示。弧的部分通过参数 θ 的范围来定义。Arc2D类可以用三种方法来封闭一个弧，它们分别是OPEN、CHORD和PIE。例如，下列代码构造了一个参数范围是 $\theta = 30$ 度到 $\theta = 75$ 度的弧，弧的封闭方式是PIE。

```
Arc2D arc=new Arc2D.Float(20f, 30f, 100f, 80f, 30f,75f, Arc2D.PIE);
```


注意 θ 角指定的是度数而非弧度。任意一点的参数 θ 通常与对应的径向角不符。例如，当 $\theta=45^\circ$ 时，从椭圆中心到椭圆上这一点的连线，恰好对应于外接矩形的对角线，如图2-11所示，显然这条直线的角度不是 45° ，除非这个椭圆恰好是一个圆。

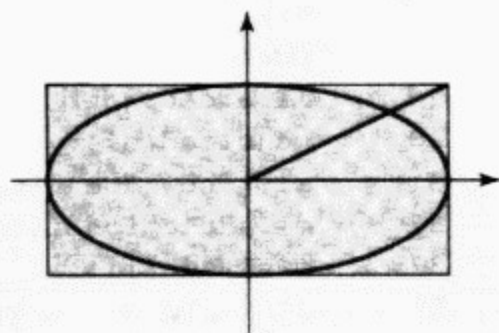


图2-11 椭圆上一点的径向直线的几何角度未必与该点的参数角度相等

采用Rectangle2D类、RoundRectangle2D类、Ellipse2D类和Arc2D类所构造的对象，它们的外接矩形都平行于x轴和y轴。但是，这不是一个严重的限制，因为只要通过恰当的几何变换，就可以很容易地得到这些对象的一般性的“旋转”版本。关于几何变换的相关内容，将在下一章中进行详细讨论。

Polygon类与rectangle类的情形很相似，它同样衍生于较早的AWT，仅支持整型坐标。通过下面的构造函数，可以完成对Polygon对象的构造：

```
Polygon (int[] xcoords, int[] ycoords, int npoints);
```

用两个整形数组定义多边形的顶点，第一个点和最后一个点连接起来，构成一个封闭路径。

2.6.2 实例

程序清单2-3给出了一个交互式的绘制程序。它允许用户绘制Java 2D的各种几何体，其中包括矩形、圆角矩形、椭圆形、弧线、线条、二次曲线、三次曲线和多边形（如图2-12所示）等。用户可以通过菜单来选择绘制的形状，并通过拖动鼠标在屏幕上绘制特定的形状。绘制效果具有持久性，也就是说，当窗口被刷新时，所绘制的图形不会消失。

43

程序清单2-3 DrawShapes.java

```
1 package chapter2;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.event.*;
6 import java.util.*;
7 import javax.swing.*;
8 //定义DrawShapes类，继承自JApplet类，实现ActionListener接口
9 public class DrawShapes extends JApplet implements ActionListener {
10     public static void main(String s[]) {
11         JFrame frame = new JFrame(); //创建主窗口
12         frame.setTitle("Drawing Geometric Shapes"); //设置标题栏
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14         JApplet applet = new DrawShapes();//生成DrawShapes实例并初始化
15         applet.init();
16         frame.getContentPane().add(applet); //将DrawShapes加入frame内容窗格
17         frame.pack();
18         frame.setVisible(true); //设置窗口可见
19     }
20
21     JavaDraw2DPanel panel = null; // JavaDraw2Dpanel实例变量
22     //重写初始化函数
23     public void init() {
24         JMenuBar mb = new JMenuBar(); //设置菜单栏
```



```
25     setJMenuBar(mb);
26     JMenu menu = new JMenu("Objects");
27     mb.add(menu);
28     JMenuItem mi = new JMenuItem("Rectangle");
29     mi.addActionListener(this);
30     menu.add(mi);
31     mi = new JMenuItem("RoundRectangle");
32     mi.addActionListener(this);
33     menu.add(mi);
34     mi = new JMenuItem("Ellipse");
35     mi.addActionListener(this);
36     menu.add(mi);
37     mi = new JMenuItem("Arc");
38     mi.addActionListener(this);
39     menu.add(mi);
40     mi = new JMenuItem("Line");
41     mi.addActionListener(this);
42     menu.add(mi);
43     mi = new JMenuItem("QuadCurve");
44     mi.addActionListener(this);
45     menu.add(mi);
46     mi = new JMenuItem("CubicCurve");
47     mi.addActionListener(this);
48     menu.add(mi);
49     mi = new JMenuItem("Polygon");
50     mi.addActionListener(this);
51     menu.add(mi);
52     panel = new JavaDraw2DPanel(); // 初始化panel并加入DrawShapes实例
53     getContentPane().add(panel);
54 }
55 // 事件响应函数
56 public void actionPerformed(ActionEvent ev) {
57     String command = ev.getActionCommand(); // 获取事件类型并做相应处理
58     if ("Rectangle".equals(command)) {
59         panel.shapeType = panel.RECTANGLE;
60     } else if ("RoundRectangle".equals(command)) {
61         panel.shapeType = panel.ROUNDRECTANGLE2D;
62     } else if ("Ellipse".equals(command)) {
63         panel.shapeType = panel.ELLIPSE2D;
64     } else if ("Arc".equals(command)) {
65         panel.shapeType = panel.ARC2D;
66     } else if ("Line".equals(command)) {
67         panel.shapeType = panel.LINE2D;
68     } else if ("QuadCurve".equals(command)) {
69         panel.shapeType = panel.QUADCURVE2D;
70     } else if ("CubicCurve".equals(command)) {
71         panel.shapeType = panel.CUBICCURVE2D;
72     } else if ("Polygon".equals(command)) {
73         panel.shapeType = panel.POLYGON;
74     }
75 }
76 }
77 // 定义JavaDraw2Dpanel类, 继承自Jpanel类, 实现鼠标事件侦听器
78 class JavaDraw2DPanel extends JPanel
```

```
79 implements MouseListener, MouseMotionListener {
80 private Vector shapes = new Vector();//定义并初始化存放形状的Vector变量
81 static final int RECTANGLE = 0;
82 static final int ROUNDRECTANGLE2D = 1;
83 static final int ELLIPSE2D = 2;
84 static final int ARC2D = 3;
85 static final int LINE2D = 4;
86 static final int QUADCURVE2D = 5;
87 static final int CUBICCURVE2D = 6;
88 static final int POLYGON = 7;
89 static final int GENERAL = 8;
90 static final int AREA = 9;
91
92 int shapeType = RECTANGLE;
93 //定义并初始化存放输入点的向量
94 Vector points = new Vector();
95 int pointIndex = 0;
96 Shape partialShape = null;
97 Point p = null;
98 //构造函数
99 public JavaDraw2DPanel() {
100     super();
101     setBackground(Color.white);//设置背景颜色为白色
102     setPreferredSize(new Dimension(640, 480));//设置窗口首选大小
103     addMouseListener(this);//加入鼠标事件侦听器
104     addMouseMotionListener(this);
105 }
106 //重写组件绘制方法
107 public void paintComponent(Graphics g) {
108     super.paintComponent(g);
109     Graphics2D g2 = (Graphics2D)g; //强制转换g作为Graphics2D对象使用
110     for (int i = 0; i < shapes.size(); i++) { //依次绘制存储的形状
111         Shape s = (Shape)shapes.get(i);
112         g2.draw(s);
113     }
114 }
115 public void mouseClicked(MouseEvent ev) { //响应单击鼠标事件
116 }
117
118 public void mouseEntered(MouseEvent ev) { //响应鼠标进入事件
119 }
120
121 public void mouseExited(MouseEvent ev) { //响应鼠标退出事件
122 }
123
124 public void mousePressed(MouseEvent ev) { //响应按下鼠标事件
125     points.add(ev.getPoint());//增加点
126     pointIndex++;
127     p = null;
128 }
129
130 public void mouseReleased(MouseEvent ev) { //响应释放鼠标事件
131     Graphics g = getGraphics();
132     Point p1 = (Point)(points.get(pointIndex-1));
```



```

133     p = ev.getPoint();//保存当前鼠标所在点
134     Shape s = null;//Shape类中的S对象设为空
135     switch (shapeType) { //根据选择的形状类型进行处理
136         case RECTANGLE://类型为矩形
137             s = new Rectangle(pl.x, pl.y, p.x-pl.x, p.y-pl.y);
138             break;
139         case ROUNDRECTANGLE2D://类型为圆角矩形
140             s = new RoundRectangle2D.Float(pl.x, pl.y,
141                 p.x-pl.x, p.y-pl.y, 10, 10); //
142             break;
143         case ELLIPSE2D://类型为椭圆
144             s = new Ellipse2D.Float(pl.x, pl.y, p.x-pl.x, p.y-pl.y);
145             break;
146         case ARC2D://类型为弧形
147             s = new Arc2D.Float(pl.x, pl.y, p.x-pl.x,
148                 p.y-pl.y, 30, 120, Arc2D.OPEN);
149             break;
150         case LINE2D://类型为直线
151             s = new Line2D.Float(pl.x, pl.y, p.x, p.y);
152             break;
153         case QUADCURVE2D://类型为二次曲线
154             if (pointIndex > 1) {
155                 Point p2 = (Point)points.get(0);
156                 s = new QuadCurve2D.Float(p2.x, p2.y, pl.x, pl.y, p.x, p.y);
157             }
158             break;
159         case CUBICCURVE2D://类型为三次曲线
160             if (pointIndex > 2) {
161                 Point p2 = (Point)points.get(pointIndex-2);
162                 Point p3 = (Point)points.get(pointIndex-3);
163                 s = new CubicCurve2D.Float(p3.x, p3.y, p2.x, p2.y,
164                     pl.x, pl.y, p.x, p.y);
165             }
166             break;
167         case POLYGON://类型为多边形
168             if (ev.isShiftDown()) {
169                 s = new Polygon();
170                 for (int i = 0; i < pointIndex; i++)
171                     ((Polygon)s).addPoint(((Point)points.get(i)).x,
172                         ((Point)points.get(i)).y);
173                 ((Polygon)s).addPoint(p.x, p.y);
174             }
175
176     }
177     if (s != null) { //如果生成了正确的形状,则存储并重绘窗口
178         shapes.add(s);
179         points.clear();
180         pointIndex = 0;
181         p = null;
182         repaint();
183     }
184 }
185
186 public void mouseMoved(MouseEvent ev) { //响应移动鼠标事件
187 }
188

```

```
189 public void mouseDragged(MouseEvent ev) { //响应拖动鼠标事件
190     Graphics2D g = (Graphics2D) getGraphics();
191     g.setXORMode(Color.white); //设置颜色
192     Point p1 = (Point) points.get(pointIndex-1);
193     switch (shapeType) { //根据所选形状类型进行绘制
194         case RECTANGLE: //绘制矩形
195             if (p != null) g.drawRect(p1.x, p1.y, p.x-p1.x, p.y-p1.y);
196             p = ev.getPoint();
197             g.drawRect(p1.x, p1.y, p.x-p1.x, p.y-p1.y);
198             break;
199         case ROUNDRECTANGLE2D: //绘制圆角矩形
200             if (p != null) g.drawRoundRect(p1.x, p1.y,
201                 p.x-p1.x, p.y-p1.y, 10, 10);
202             p = ev.getPoint();
203             g.drawRoundRect(p1.x, p1.y, p.x-p1.x, p.y-p1.y, 10, 10);
204             break;
205         case ELLIPSE2D: //绘制椭圆形
206             if (p != null) g.drawOval(p1.x, p1.y, p.x-p1.x, p.y-p1.y);
207             p = ev.getPoint();
208             g.drawOval(p1.x, p1.y, p.x-p1.x, p.y-p1.y);
209             break;
210         case ARC2D: //绘制弧形
211             if (p != null) g.drawArc(p1.x, p1.y, p.x-p1.x, p.y-p1.y, 30, 120);
212             p = ev.getPoint();
213             g.drawArc(p1.x, p1.y, p.x-p1.x, p.y-p1.y, 30, 120);
214             break;
215         case LINE2D: //绘制直线形，与多边形处理相同
216         case POLYGON: //绘制多边形
217             if (p != null) g.drawLine(p1.x, p1.y, p.x, p.y);
218             p = ev.getPoint();
219             g.drawLine(p1.x, p1.y, p.x, p.y);
220             break;
221         case QUADCURVE2D: //绘制二次曲线
222             if (pointIndex == 1) {
223                 if (p != null) g.drawLine(p1.x, p1.y, p.x, p.y);
224                 p = ev.getPoint();
225                 g.drawLine(p1.x, p1.y, p.x, p.y);
226             } else {
227                 Point p2 = (Point) points.get(pointIndex-2);
228                 if (p != null) g.draw(partialShape);
229                 p = ev.getPoint();
230                 partialShape = new QuadCurve2D.Float(p2.x, p2.y,
231                     p1.x, p1.y, p.x, p.y);
232                 g.draw(partialShape);
233             }
234             break;
235         case CUBICCURVE2D: //绘制三次曲线
236             if (pointIndex == 1) {
237                 if (p != null) g.drawLine(p1.x, p1.y, p.x, p.y);
238                 p = ev.getPoint();
239                 g.drawLine(p1.x, p1.y, p.x, p.y);
240             } else if (pointIndex == 2) {
241                 Point p2 = (Point) points.get(pointIndex-2);
242                 if (p != null) g.draw(partialShape);
243                 p = ev.getPoint();
244                 partialShape = new QuadCurve2D.Float(p2.x, p2.y,
```



```

245         p1.x, p1.y, p.x, p.y);
246         g.draw(partialShape);
247     } else {
248         Point p2 = (Point)points.get(pointIndex-2);
249         Point p3 = (Point)points.get(pointIndex-3);
250         if (p != null) g.draw(partialShape);
251         p = ev.getPoint();
252         partialShape = new CubicCurve2D.Float(p3.x, p3.y,
253         p2.x, p2.y, p1.x, p1.y, p.x, p.y);
254         g.draw(partialShape);
255     }
256     break;
257 }
258 }
259 }

```

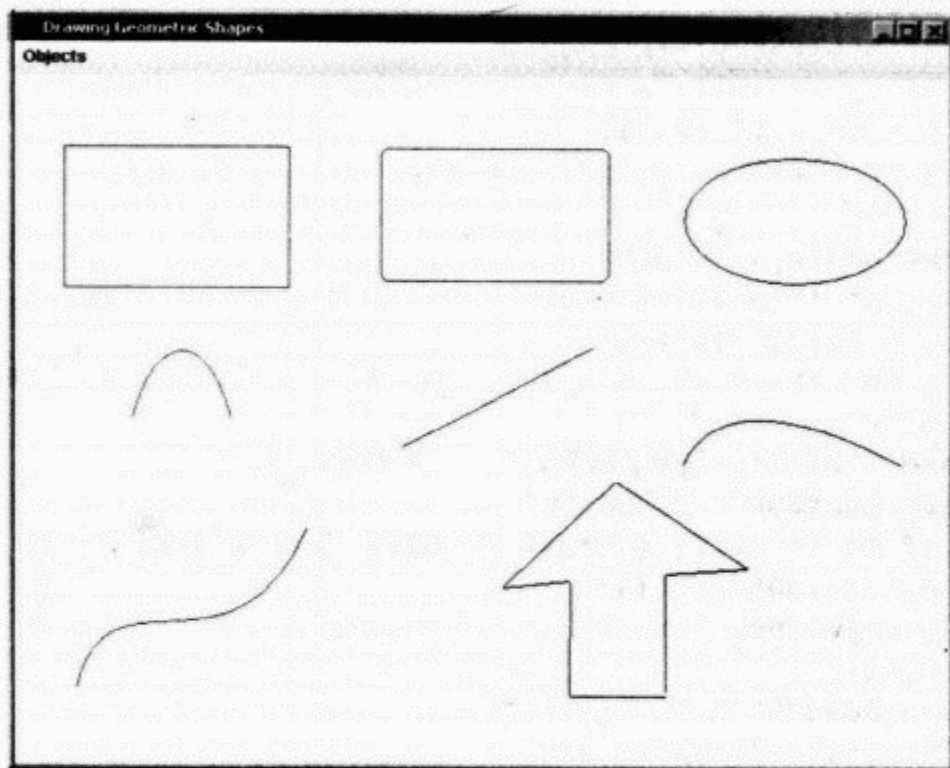


图2-12 绘制用Java 2D定义的基本几何体

这是一个简单的绘制程序，它不能够支持所有的特性，如不支持文件的I/O，但它包括一个典型绘制包的主要特性，它允许用户通过鼠标绘制任意数量的基本形状。

DrawShapes类用于定义一个带有菜单的applet，菜单的选项可用来选择所绘制的形状类型。JavaDraw2DPanel是JPanel的一个子类，用来实现实际的绘制逻辑。变量panel是JavaDraw2DPanel类的实例（第52行），菜单的选择由DrawShapes类来处理，对事件的响应把panel中的shapeType变量设置为被选中的形状类型。

在JavaDraw2DPanel类中，包含一个vector容器shapes，用来存储用户绘制的所有形状。当一个特定的图形被绘制完毕后，将其加入到容器中。因为Vector类被定义为动态数据结构，形状的数量是不受限制的。paintComponent方法也十分简单，它遍历vector容器，不需要考虑各个形状的确切类型，通过调用每个形状的draw方法，来完成每个形状的绘制（第110~112行）。之所以能以这种优雅的方式完成绘制，是因为Java语言具有强大的多态支持功能及Graphics2D类。

JavaDraw2DPanel类通过处理鼠标事件来实现绘制功能，绘制形状的细节取决于形状的类型，矩形、圆角矩形、椭圆、弧线和线条都是通过它们的外接矩形来定义的。绘制是通过拖动鼠标来完成的，鼠标的拖动路线是从外接矩形的一角到其对角。二次曲线是通过三个控制点来定义，它的绘制是通过两次拖动鼠标，定义两条线段来完成的。三次曲线是通过四个控制点来

定义的，通过三次鼠标拖动来完成的。多边形允许有任意多个点，它的绘制通过反复的拖动鼠标来完成，双击鼠标将终止多边形的构造。

在构建一个形状时，“橡皮条”的实现提供了一个视觉线索。当用户拖动鼠标时，对应于当前鼠标位置的形状将被显示出来，并且不断地更新。它是通过在鼠标事件处理器中用XOR绘制模式来完成的。当鼠标键按下时，当前鼠标的位置被保存在vector容器points中（第125行）。当鼠标被拖动时，将获取Graphics2D对象（第190行），通过调用setXORMode方法设置其XOR模式（第191行）。基于形状的类型和当前鼠标位置，可以绘制出一个临时形状，后续的调用会擦掉这个临时形状，并代之以一个新的临时形状，这就产生了一个动态的“橡皮条”效应。在XOR绘制模式下，对同一形状的第二次的绘制，会擦除第一次的绘制结果。当鼠标键松开时（第130行），创建一个新点。如果对当前的形状类型来说有足够多的点，就可以构建一个新的完整的形状，并将其添加到shapes容器中。如果对当前的形状类型来说点的数量不足，就定义一个不完整的形状。

2.7 构造区域几何模型

生成更复杂的形状的一种方法，是通过组合已存在的几种形状而得到新的形状，这就是所谓的构造区域几何模型。Area类就是为了完成构造区域几何模型而设计的，它支持四个操作：合并、交叉、差分和对称差分。这些集合论方面的操作，都是针对两个区域进行的，并生成一个新的区域。两个区域合并操作的结果是，将两个区域的所有点包括进来而形成整体。两个区域交叉操作的结果是，只包含同时属于两个区域的那些点。两个区域的差分操作，将属于第一个区域但不属于第二个区域的所有点保留下来。两个区域的对称差分包含所有严格属于两个区域之一的那些点。Area对象可以采用下面的构造函数，从任何Shape对象进行构造：

Area (Shape s)

四种操作都可以通过调用Area对象的方法来完成，调用形式如下：

```
void add (Area a) //将指定区域的形状添加到此区域的形状中
void intersect (Area a) //将此区域的形状设置为当前形状与指定区域形状的交集
void subtract (Area a) //从此区域的形状中减去指定区域的形状
void exclusiveOr (Area a) //将此区域的形状设置为其当前形状与指定区域形状的并集，减去二者的交集
```

操作结果存放在当前Area对象中，第二个Area对象作为方法的参数，不会因操作而改变。

程序清单2-4显示了构造区域几何模型的四种种区域操作的效果。两个区域形状运用这四种操作获得四个新的形状（如图2-13所示）。

49

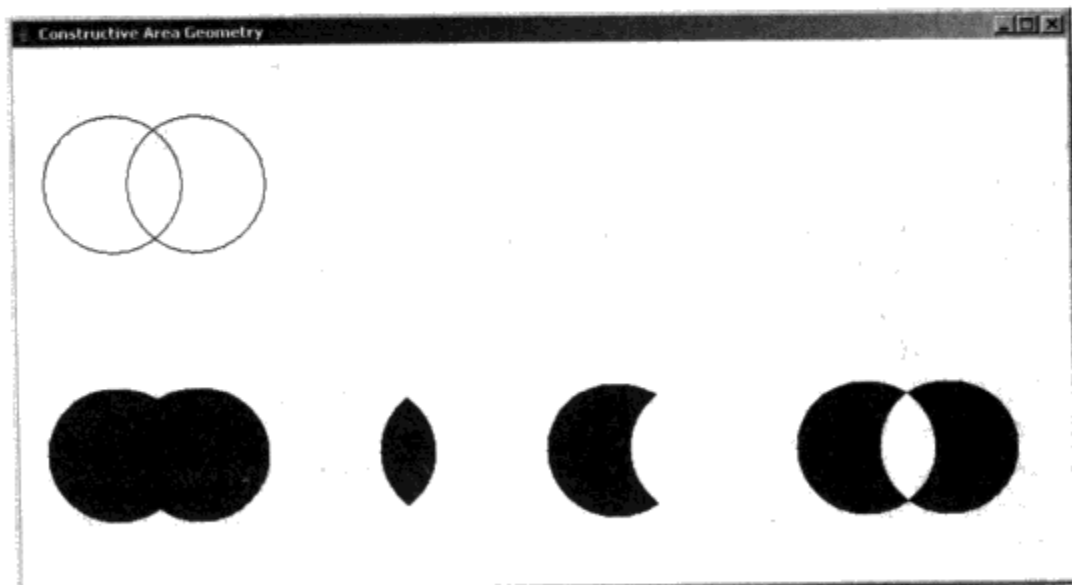


图2-13 第一行为两个区域的形状对象。第二行为四种区域操作结果。
四种区域操作分别是：add、intersect、subtract、exclusiveOr

程序清单2-4 AreaGeomtry.java

```
1 package chapter2;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.geom.*;
7 //定义AreaGeometry类, 继承自JApplet类
8 public class AreaGeometry extends JApplet {
9     public static void main(String s[]) {
10         JFrame frame = new JFrame();           //创建主窗口
11         frame.setTitle("Constructive Area Geometry");//设置标题栏
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         JApplet applet = new AreaGeometry();    //生成AreaGeometry实例并初始化
14         applet.init();
15         frame.getContentPane().add(applet);      //将AreaGeometry实例加入frame内容窗格
16         frame.pack();
17         frame.setVisible(true);                 //设置窗口可见
18     }
19     //重写初始化函数
20     public void init() {
21         JPanel panel = new AreaPanel();
22         getContentPane().add(panel);
23     }
24 }
25 //定义AreaPanel类, 继承自JPanel类
26 class AreaPanel extends JPanel {
27     public AreaPanel() {
28         setPreferredSize(new Dimension(760, 400)); //设置窗口首选大小
29     }
30     //重写组件绘制方法
31     public void paintComponent(Graphics g) {
32         Graphics2D g2 = (Graphics2D)g;
33         Shape s1 = new Ellipse2D.Double(0, 0, 100, 100); //定义两个椭圆形状
34         Shape s2 = new Ellipse2D.Double(60, 0, 100, 100);
35         Area a1;
36         Area a2 = new Area(s2);                //根据s2创建Area对象a2
37         g2.translate(20, 50);                  //进行平移
38         g2.draw(s1);                           //绘制s1
39         g2.draw(s2);                           //绘制s2
40         g2.translate(0, 200);                  //进行平移
41         a1 = new Area(s1);                     //根据s1创建Area对象a1
42         a1.add(a2);                            //将a2区域添加到a1区域中
43         g2.fill(a1);                           //填充a1
44         g2.translate(180, 0);                  //进行平移
45         a1 = new Area(s1);                     //根据s1重新初始化a1
46         //将a1区域的形状设置为其当前a2区域的形状与a1区域的形状的交集
47         a1.intersect(a2);
48         g2.fill(a1);                           //填充新区域
49         g2.translate(180, 0);                  //进行平移
50         a1 = new Area(s1);                     //根据s1重新初始化a1
51         a1.subtract(a2);                       //从a2区域的形状中减去a1区域的形状
52         g2.fill(a1);                           //填充新区域
53         g2.translate(180, 0);                  //进行平移
54         a1 = new Area(s1);                     //根据s1重新初始化a1
55         a1.exclusiveOr(a2);
```

```

        // 将a1区域的形状设置为其a2区域的形状与a1区域形状的合并区域，并减去其交集
55      g2.fill(a1);                                //填充新区域
56    }
57  }

```

50

将两个重叠的圆S1和S2定义为两个原始的形状，它们显示在屏幕上方。Area对象a1和a2根据这两个形状进行创建（第35~36行）。通过调用add、intersect、subtract和exclusiveOr方法，实现对a1和a2的这四种操作。最终的结果以填充的形状显示在屏幕下方（第42~55行）。由于Area类实现了Shape接口，因此Area对象可以直接传递给Graphics2D对象的fill方法，从而完成填充效果。

2.8 一般路径

Graphics2D引擎内部通过五种基本的曲线段或操作，完成任意形状的边界绘制。Shape接口提供的方法可以获得一个PathIterator接口，它用五种类型的曲线段来描述形状边界的路径。PathIterator定义了五个曲线段常量：

```

SEG_MOVETO
SEG_LINETO
SEG_QUADTO
SEG_CUBICTO
SEG_CLOSE

```

构造自定义形状的一种强大方法是使用GeneralPath类，它直接支持用Graphics2D所知的5种基本曲线段来构建路径。GeneralPath类的如下方法，分别实现对这五种线段类型的构建：

```

void moveTo(float x, float y); //通过移动到指定的坐标（以 float 精度指定），将一个点添加到
    路径中。
void lineTo(float x, float y); //通过绘制一条从当前坐标到指定新坐标（以 float 精度指定）的
    直线，将一个点添加到路径中。
void quadTo(float x1, float y1, float x2, float y2); //通过绘制与当前坐标和指定坐标
    (x2,y2)都相交的二次曲线，并将指定点(x1,y1)用做二次曲线参数控点，可以将由两个新点定义的曲
    线段添加到路径中。
void curveTo(float x1, float y1, float x2, float y2, float x3, float y3); //通过绘
    制与当前坐标和指定坐标(x3,y3)都相交的贝塞尔曲线，并将指定点(x1,y1)和 (x2,y2) 用做贝塞尔
    曲线的控点，可以将由三个新点定义的曲线段添加到路径中。
void closePath(); //通过绘制一条向后延伸到最后一个moveTo的坐标的直线，封闭当前子路径。如果该
    路径已封闭，则此方法无效。

```

路径的构造过程可以视为是通过一支“笔”进行的绘制过程。在任意时刻，这支笔都有一个“当前位置”。通过moveTo方法将笔移动到一个新的位置(x, y)处，而不绘制任何东西。lineTo方法从当前位置到点(x, y)处绘制一条直线，并以这个新的点作为笔的当前位置。quadTo方法绘制一条二次曲线，起点为当前位置，终点为(x2, y2)，(x1, y1)是曲线的中间控点。CurveTo方法绘制一条三次曲线，起点为当前位置，终点为(x3, y3)，(x1, y1)和(x2, y2)是曲线中间的两个控点。closePath方法绘制一条直线，连到由上一次调用moveTo方法所定义的点。

下列代码段所构造的图形如图2-14所示。

```

GeneralPath path=new GeneralPath();
path.moveTo(-2f, 0f);
path.quadTo(0f, 2f, 2f, 0f);
path.quadTo(0f, -2f, -2f, 0f);
path.moveTo(-1f, 0.5f);
path.lineTo(-1f, 0.5f);

```

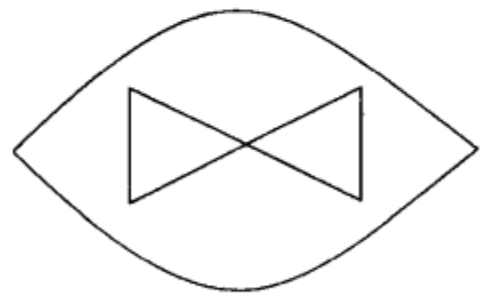


图2-14 通过GeneralPath对象定义的形状

51


```

path.lineTo(1f, 0.5f);
path.lineTo(1f, -0.5f);
path.closePath();

```

路径定义了形状的边界或轮廓。但是，为了完整地定义一个形状，必须指定形状的内部和外部分别是什么。例如对一个形状进行填充时，这类信息是需要的。由于路径可以通过具有复杂关系的多个区域来构造，因此内部区域的问题并非总是简单的。缠绕规则（winding rules）用来定义在什么条件下某个区域属于形状内部。PathIterator接口定义的两个缠绕规则是：

```

WIND_EVEN_ODD
WIND_NON_ZERO

```

要确定由路径形成的某一区域是否属于形状的内部，可以画一条穿过该区域的直线，考虑从外部到此区域这条直线与路径相交的次数。奇偶规则（even-odd rule）规定，如果相交次数为奇数，则该区域在内部，如果相交次数为偶数，则该区域在外部。实质上，路径是用来区分内部和外部的边界线，因此这条直线每穿过一次路径，属于内部还是外部区域的情况就改变一次。

非零规则（nonzero rule）规定，要考虑直线穿过路径的方向，相交的次数可以是正的或负的。在面向路径的方向上，如果直线从左向右穿过路径，则相交的次数加一，否则相交次数减一。如果相交次数非零，则区域在内部。非零规则的实质是，沿着路径方向移动时，路径的左面被定义为内部，右面被定义为外部。

如图2-15所示的路径由两个正方形组成。采用奇偶规则进行判定时，里面的正方形区域被判定为“外部”，这是因为相交次数为偶数。采用非零规则判定时，如果路径的方向如图所示，那么这个内部正方形区域将被判定为“内部”，这是因为穿过路径的次数非零。



图2-15 采用奇偶规则和非零规则对内部区域进行定义

另外一个例子如图2-16所示。采用奇偶规则进行判定时，在区域里面的两个三角形被判定为“外部”，它们是曲线区域里面的两个洞。但是，当采用非零规则判定时，只有一个三角形被判定为空洞。右侧的三角形被判定为内部，因为它的路径方向和外层的路径方向相同，当直线穿过两条路径时，相交的次数并未相互抵消。

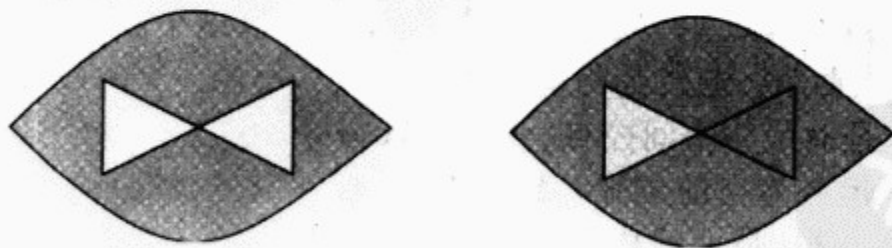


图2-16 左图：奇偶规则。右图：非零规则

程序清单2-5给出了一个运用GeneraPath类构造形状的程序实例及缠绕规则的效果。该实例产生了两个形状，并以三种不同风格进行显示，这三种风格分别是：只显示路径、运用奇偶规则显示判定结果和运用非零规则显示判定结果（如图2-17所示）。

程序清单2-5 GustomPath.java

```

1 package chapter2;
2

```



```
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.geom.*;
7 //定义CustomPath类, 继承自JApplet类
8 public class CustomPath extends JApplet {
9     public static void main(String s[]) {
10         JFrame frame = new JFrame();           //创建主窗口
11         frame.setTitle("GeneralPath and Winding Rules");//设置窗口标题
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         JApplet applet = new CustomPath();      //生成CustomPath实例并初始化
14         applet.init();
15         frame.getContentPane().add(applet);      //将CustomPath实例加入frame内容窗格
16         frame.pack();
17         frame.setVisible(true);                 //设置窗口可见
18     }
19     //重写初始化函数
20     public void init() {
21         JPanel panel = new PathPanel();
22         getContentPane().add(panel);
23     }
24 }
25 //定义PathPanel类, 继承自JPanel类
26 class PathPanel extends JPanel {
27     public PathPanel() {
28         setPreferredSize(new Dimension(640, 480)); //设置窗口首选大小
29     }
30
31     public void paintComponent(Graphics g) {
32         super.paintComponent(g);
33         Graphics2D g2 = (Graphics2D)g;         //强制转换g并作为Graphics2D对象使用
34         //按奇偶规则创建一般路径对象
35         GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
36         //设置变量值
37         float x0 = 1.0f;
38         float y0 = 0.0f;
39         float x1 = (float)Math.cos(2*Math.PI/5.0);
40         float y1 = (float)Math.sin(2*Math.PI/5.0);
41         float x2 = (float)Math.cos(4*Math.PI/5.0);
42         float y2 = (float)Math.sin(4*Math.PI/5.0);
43         float x3 = (float)Math.cos(6*Math.PI/5.0);
44         float y3 = (float)Math.sin(6*Math.PI/5.0);
45         float x4 = (float)Math.cos(8*Math.PI/5.0);
46         float y4 = (float)Math.sin(8*Math.PI/5.0);
47         path.moveTo(x2,y2); //移动到(x2,y2)处, 将该点添加到路径中
48         path.lineTo(x0,y0); //绘制从当前坐标到(x0,y0)的直线, 将该点添加到路径中
49         path.lineTo(x3,y3); //绘制从当前坐标到(x3,y3)的直线, 将该点添加到路径中
50         path.lineTo(x1,y1); //绘制从当前坐标到(x1,y1)的直线, 将该点添加到路径中
51         path.lineTo(x4,y4); //绘制从当前坐标到(x4,y4)的直线, 将该点添加到路径中
52         path.closePath(); //封闭当前子路径
53         AffineTransform tr = new AffineTransform(); //创建仿射变换模型
54         tr.setToScale(100,100); //进行缩放
55         g2.translate(120,120); //进行平移
56         path = (GeneralPath)tr.createTransformedShape(path); //创建路径
57         g2.draw(path); //绘制路径
58         g2.translate(200,0); //进行平移
59         g2.fill(path); //填充路径
```



```

58 path.setWindingRule(GeneralPath.WIND_NON_ZERO); //设定为非零规则
59 g2.translate(200,0); //进行平移
60 g2.fill(path); //填充路径
61
62 path.reset(); //重置路径
63 path.moveTo(x0, y0); //移动到(x0, y0)处, 将该点添加到路径中
64 path.lineTo(x1, y1); //绘制从当前坐标到(x1, y1)的直线, 将该点添加到路径中
65 path.lineTo(x2, y2); //绘制从当前坐标到(x2, y2)的直线, 将该点添加到路径中
66 path.lineTo(x3, y3); //绘制从当前坐标到(x3, y3)的直线, 将该点添加到路径中
67 path.lineTo(x4, y4); //绘制从当前坐标到(x4, y4)的直线, 将该点添加到路径中
68 path.closePath(); //封闭当前子路径
69 path.moveTo(x0, y0); //移动到(x0,y0)处, 将该点添加到路径中
70 path.quadTo(x4, y4, x1, y1);
71 path.quadTo(x2, y2, x3, y3);
72 path.closePath(); //封闭当前子路径
73 path.moveTo(x4, y4); //移动到(x4,y4)处, 将该点添加到路径中
74 path.curveTo(x1, y1, x3, y3, x2, y2); //将由两个新点定义的曲线段添加到路径中
75 path.curveTo(x1, y1, x3, y3, x4, y4); //将由两个新点定义的曲线段添加到路径中
76 path = (GeneralPath)tr.createTransformedShape(path); //创建路径
77 g2.translate(-400,220); //进行平移
78 g2.draw(path); //绘制路径
79 path.setWindingRule(GeneralPath.WIND_EVEN_ODD); //设定为奇偶规则
80 g2.translate(200,0); //进行平移
81 g2.fill(path); //填充路径
82 path.setWindingRule(GeneralPath.WIND_NON_ZERO); //设定为非零规则
83 g2.translate(200,0); //进行平移
84 g2.fill(path); //填充路径
85 }
86 }

```

54

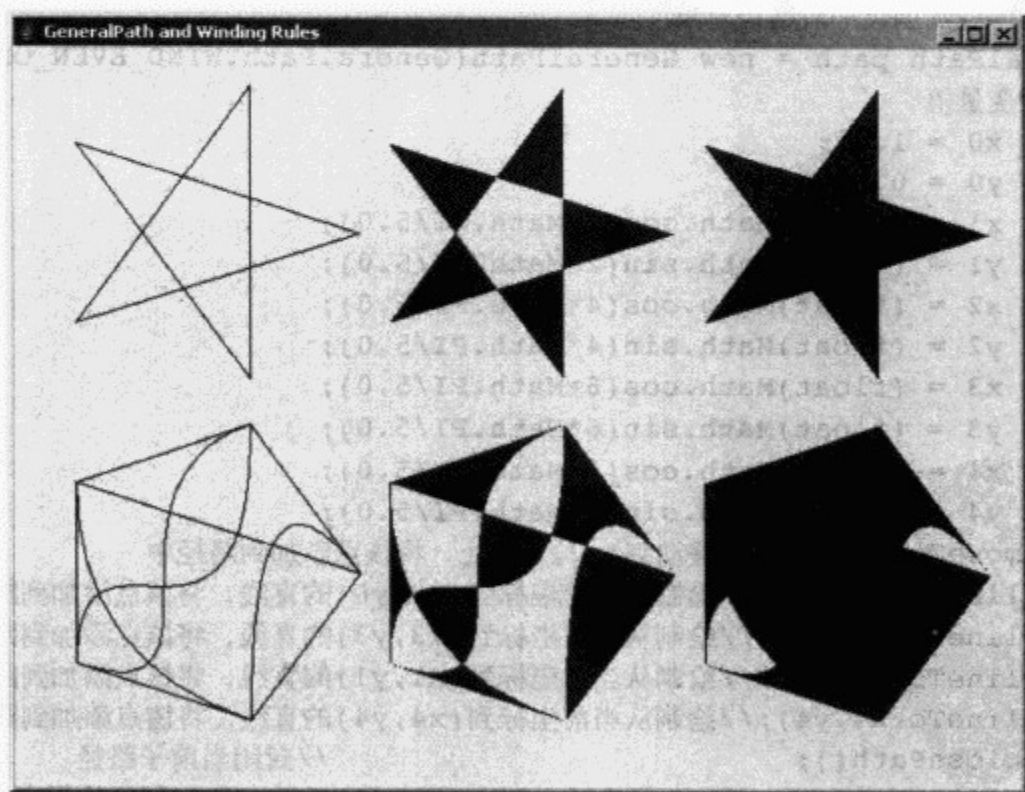


图2-17 一般路径和由奇偶规则与非零规则根据该路径形成的区域

两个形状通过GeneralPath类进行构造。五角星完全是通过线段来建立的。将一个圆进行五平分, 圆上的平分点就是五角星的顶点(第35~44行)。GeneralPath对象的reset方法清除对象的路径(第62行)。另外一个形状是一个五边形, 它由一些线段、二次曲线和三次曲线组成。每

个形状显示为三个不同的版本。首先，通过调用draw方法来绘制形状的轮廓路径（第55、78行）。然后，将形状的缠绕规则设为按奇偶规则进行判定，再通过调用fill方法来显示填充内部区域的形状（第57、81行）。最后，把形状的缠绕规则设为按非零规则进行判定，再次通过调用fill方法来显示填充内部区域的形状（第60、84行）。可以清楚地看到，非零规则通常要比奇偶规则产生更多的内部区域。

主要的类和方法

- **javax.swing.JComponent.paintComponent(Graphics)** 该方法通常需要重写，以进行组件的自定义绘制。
- **java.awt.Graphics2D** 该类为Java 2D绘制引擎提供一个接口。
- **java.awt.Graphics2D.draw(Shape)** 该方法负责绘制形状轮廓。
- **java.awt.Graphics2D.fill(Shape)** 该方法可以对形状内部进行填充。
- **java.awt.Shape** 2D几何体的接口。
- **java.awt.geom.GeneralPath** 该类用Shape接口定义的各种线段类型来定义任意的2D轮廓。
- **java.awt.geom.Area** 该类用来构造区域几何模型。

55

关键术语

- **坐标系 (coordinate system)** 将几何点与有序数元组的代数量进行关联的方法。
- **参数方程 (parametric equation)** 坐标变量通过参数函数形式来表达的一组方程。
- **世界空间 (world space)** 图形模型的公用参考坐标空间。
- **对象空间 (object space)** 与单个对象关联的局部坐标空间。
- **设备空间 (device space)** 面向特定输出设备的坐标空间。
- **构造区域几何模型 (constructive area geometry)** 通过对现有的形状区域进行合并及交叉之类的集合操作，从而生成新几何体的方法。
- **缠绕规则 (winding rule)** 确定由给定轮廓形成的内部区域的规则。

本章提要

- 这一章对2D图形系统的基本原理和Java 2D编程进行了介绍。一个典型的2D图形绘制流水线包含三个不同的坐标空间。在对象空间中，各种可见对象定义在便于对象建模的局部坐标系里。通过对象变换，对象空间能够映射到世界空间上，世界空间是所有对象的公用参考坐标空间。在世界空间中建模的图形场景，要映射到设备空间后才可以显示。从世界空间到设备空间的变换称为观察变换。
- 2D曲线通常用方程来表示。参数方程对于图形应用来说，显得非常方便。
- 可视对象的几何模型的建立，通常是通过基本元素和构造操作来完成的。Java 2D提供了一组丰富的几何基元，它们包括线、矩形、圆角矩形、椭圆、弧线、二次曲线和三次曲线。Shape接口提供了用于几何描述的基本框架。
- 构造区域几何模型是针对已有区域进行操作来生成新形状的一种技术。Java 2D为构造区域几何模型提供了四种集合论操作。
- 在Shape 接口中，可以用五种不同线段的基本操作来定义路径。GeneralPath类提供了可以直接访问的五种线段类型。有两种缠绕规则可以用来确定区域的内部。

复习题

2.1 对象变换和视图变换有什么不同？

2.2 在2D坐标系中标注下面的点：(1,3)，(-2,1.5)，(0,-2)，(0,0)。

第3章 2D图形学：绘制细节

学习目标

- 理解颜色空间。
- 使用Java Color类。
- 能使用不同的涂色方式来绘制可视对象。
- 运用各种笔划。
- 构造仿射变换，包括平移、旋转、缩放、错切和反射。
- 理解对象变换和视图变换。
- 组合基本变换，构成复杂的变换。
- 明确变换的复合规则。
- 使用裁剪路径。
- 运用字体和字体度量。
- 理解字形、连体字和衍生字体。

59

3.1 引言

本章将介绍2D图形绘制的一些重要属性，以及它们在Java 2D中的实现。除了图形对象结构的几何定义以外，图形对象的其他很多属性和操作，对图像的绘制也有非常重要的作用。颜色、笔划类型、变换、复合规则、裁剪路径和绘制提示，是影响绘制结果的一些因素。

颜色和涂色样式是很突出的可视属性。颜色在颜色空间里以数字的形式进行表示。Java定义了Color类表示物体的颜色。涂色作为一个一般化的颜色概念，能够表示复杂的颜色着色模式。Java 2D使用Paint接口统一不同的涂色方式，Color、GradientPaint和TexturePaint 类都实现了Paint接口，这些类可以在绘制2D图形时使用。

笔划（stroke）定义了画笔的细节，如宽度、线帽、连接与虚线等。与Paint类似，Graphics2D允许选择一个Stroke对象作为画笔的绘制属性。具体的类BasicStroke实现了Stroke接口，并提供了一些常用的笔划设定。

变换（transformation）是计算机图形学中至关重要的一部分。变换可以用来改变几何体形状和视图。仿射变换是计算机图形学中常用的变换类型，平移、旋转、缩放、反射和错切都属于仿射变换。

颜色、透明度和重叠物体的一般合成规则也是很有趣的绘制属性。一种称为Porter-Duff规则的通用合成规则集在2D图形学中经常用到。最新版本的Java 2D完全支持Porter-Duff规则。

裁剪路径（clip path）是另外一种绘制属性，用于定义对象实际的可视区域。

文本（text）是一种特殊的图形对象，它具有很紧凑的表示形式。文本的实际几何形状由字体预先定义。Java 2D 对字体提供了广泛的支持，它除了支持各种字体的文本绘制等标准应用之外，也支持诸如获取字符轮廓等其他高级特性。

3.2 颜色和涂色

3.2.1 颜色空间

颜色是图形系统的重要属性。光的颜色是与它的波长或频率相关联的，通常光有一个频率范围，精确地描述这个频率范围相当复杂。实际上，在图形系统中，只需要定义人眼可见和可区分的颜色。颜色空间（color space）系统通常以数字的形式定义可见颜色。有一些不同的颜色空间，以精确和量化的方式描述颜色，一般的方法是选择一部分确定的基色（如红、绿、蓝），并用基色的组合表示任意一种颜色：

$$c = r \cdot p_r + g \cdot p_g + b \cdot p_b$$

系数（ r, g, b ）表示颜色 c 的红、绿、蓝颜色分量，这种方法提供了一种简便的颜色数字描述形式。

CIEXYZ是一种颜色标准，使用X、Y、Z三种基色来代替红、绿、蓝三种基色，任何可见颜色都可以表示成X、Y、Z三种基色的线性组合。然而，在物理设备上直接实现CIEXYZ通常很困难。

大多数显示器使用RGB（红、绿、蓝）颜色空间，打印机使用CMYK（青、品红、黄、黑）颜色空间，这些基色的组合表示不同的颜色。这些颜色系统虽然跟设备很贴近，但它们通常是设备相关的，并且，无法用正系数表示出所有可见颜色。

sRGB（标准RGB）是一种真正做到了设备无关的颜色空间。该颜色空间虽然使用与其他RGB系统相同的红、绿、蓝基色，但是它对颜色的定义进行了标准化，从而使这些颜色的定义是完全设备无关的。

3.2.2 颜色

构建一个几何形状之后，可以在Graphics2D中使用fill(Shape)或draw(Shape)进行绘制。为了设定对象的绘制颜色，需要使用Graphics类的如下方法：

```
void setColor(Color c) //设定颜色为c
```

一个Color对象定义了一种颜色。Color类默认情况下使用sRGB颜色空间，它是一种标准的颜色空间。一种颜色包含Red、Green、Blue和Alpha四种分量，其中Alpha定义了颜色的透明度。颜色类定义了如下一些颜色常量值：

```
black    //黑色
blue     //蓝色
cyan     //青色
darkGray //暗灰色
gray     //灰色
green    //绿色
lightGray //浅灰色
magenta  //洋红
orange   //橙色
pink     //粉红色
red      //红色
white    //白色
yellow   //黄色
```

注意 这些名字并不符合Java命名习惯，Java定义的常量应该使用大写字母。从JDK 1.4之后可以使用新的常量，有：BLACK、BLUE、CYAN、DARK_GRAY、GREEN、LIGHT_GRAY、MAGENTA、ORANGE、PINK、RED、WHITE和YELLOW。

其他颜色可以很容易地使用Color类的构造函数加以实现。在下列构造函数中，可以直接使用RGB值：

```
//使用RGB值的构造函数
Color(int r, int g, int b);
Color(int rgb);
Color(float r, float g, float b);
```

第一个构造函数使用范围在0~255之间的整数值表示三色分量。第二个构造函数与第一个构造函数类似，但是三色值被封装成一个整数值，用于表示颜色。第三个构造函数使用范围在0.0~1.0之间的浮点值定义颜色分量。

除了RGB值之外，也可以声明表示颜色透明度的alpha值。下列构造函数允许声明alpha值：

```
//带有alpha值的构造函数
Color(int r, int g, int b, int a);
Color(int rgba, boolean hasAlpha);
Color(float r, float g, float b, float a);
```

Color类的另外一个构造函数允许对颜色空间进行声明：

```
//带颜色空间的构造函数
Color(ColorSpace colorSpace, float[] components, float alpha);
```

下面的代码段绘制了三个不同颜色的正方形：

```
public void paintComponent(Graphics g){
    g.setColor(Color.red); //设定红颜色
    g.draw(new Rectangle(0,0,100,100)); //绘制矩形
    g.setColor(new Color(0,255,128));
    g.draw(new Rectangle(100,0,100,100)); //绘制颜色为(0,255,128)的矩形
    g.setColor(new Color(0.5f,0.0f,1.0f));
    g.draw(new Rectangle(200,0,100,100)); //绘制颜色为(0,255,128)的矩形
}
```

61

程序清单3-1演示了在Java 2D图形绘制中基色组合和颜色使用的情况。这个程序显示了三个相交的圆，它们分别是使用红、绿、蓝三种颜色进行绘制的。七个不同的区域表示三种颜色的不同组合结果。窗口右边的三个滑动条分别控制红、绿、蓝三种基色的值（如图3-1所示）。

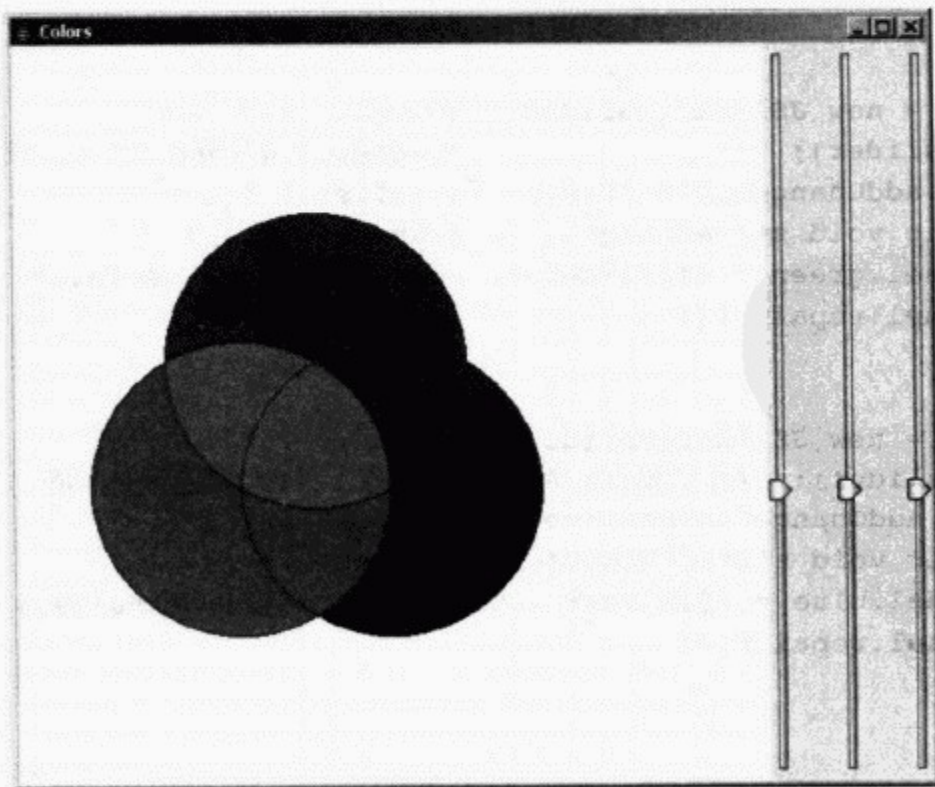


图3-1 红、绿、蓝三色的各种组合结果

程序清单3-1 TestColors.java

```

1 package chapter3;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import javax.swing.event.*;
7 import java.awt.geom.*;
8 //定义类TestColors, 继承自JApplet
9 public class TestColors extends JApplet {
10     public static void main(String s[]) {
11         JFrame frame = new JFrame();           //创建主窗口
12         frame.setTitle("Colors");              //设置窗口标题
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14         JApplet applet = new TestColors();      //创建TestColors实例
15         applet.init();
16         frame.getContentPane().add(applet);     //添加TestColors对象到主窗口
17         frame.pack();
18         frame.setVisible(true);                //设置窗口可见
19     }
20
21     ColorPanel panel;                          //声明ColorPanel变量
22     public void init() {                       //初始化函数
23         panel = new ColorPanel();              //初始化ColorPanel对象
24         Container cp = getContentPane();
25         cp.setLayout(new BorderLayout());        //设置窗口布局
26         cp.add(panel, BorderLayout.CENTER);
27         JPanel p = new JPanel();
28         cp.add(p, BorderLayout.EAST);
29         p.setLayout(new GridLayout(1,3,30,10)); //生成滚动条对象
30         JSlider slider = new JSlider(JSlider.VERTICAL,0,255,100);
31         p.add(slider);                          //添加滚动条并设置事件处理, 获取红颜色分量的值
32         slider.addChangeListener(new ChangeListener() {
33             public void stateChanged(ChangeEvent ev) {
34                 panel.red = ((JSlider)(ev.getSource())).getValue();
35                 panel.repaint();
36             }
37         });
38         slider = new JSlider(JSlider.VERTICAL,0,255,100);
39         p.add(slider);                          //添加滚动条并设置事件处理, 获取绿颜色分量的值
40         slider.addChangeListener(new ChangeListener() {
41             public void stateChanged(ChangeEvent ev) {
42                 panel.green = ((JSlider)(ev.getSource())).getValue();
43                 panel.repaint();
44             }
45         });
46         slider = new JSlider(JSlider.VERTICAL,0,255,100);
47         p.add(slider); //添加滚动条并设置事件处理, 获取蓝颜色分量的值
48         slider.addChangeListener(new ChangeListener() {
49             public void stateChanged(ChangeEvent ev) {
50                 panel.blue = ((JSlider)(ev.getSource())).getValue();
51                 panel.repaint();
52             }
53         });
54     }
55 }

```

```
56 //定义ColorPanel类, 继承自JPanel类
57 class ColorPanel extends JPanel{
58     int red = 100; //声明颜色变量, 并设置初始值
59     int green = 100;
60     int blue = 100;
61
62     public ColorPanel() {
63         . setPreferredSize(new Dimension(500, 500)); //设置组件首选大小
64         setBackground(Color.white); //设置组件背景颜色为白色
65     }
66     //重写组件绘制函数
67     public void paintComponent(Graphics g) {
68         super.paintComponent(g);
69         Graphics2D g2 = (Graphics2D)g;
70         Shape rc = new Ellipse2D.Double(100, 113, 200, 200); //创建三个椭圆形状
71         Shape gc = new Ellipse2D.Double(50, 200, 200, 200);
72         Shape bc = new Ellipse2D.Double(150, 200, 200, 200);
73         Area ra = new Area(rc); //根据上述椭圆形状创建Area对象
74         Area ga = new Area(gc);
75         Area ba = new Area(bc);
76         Area rga = new Area(rc);
77         rga.intersect(ga); //求ra和ga的交集区域
78         Area gba = new Area(gc);
79         gba.intersect(ba); //求ga和ba的交集区域
80         Area bra = new Area(bc);
81         bra.intersect(ra); //求ba和ra的交集区域
82         Area rgba = new Area(rga);
83         rgba.intersect(ba); //求ra、ga和ba的交集区域
84         ra.subtract(rga);
85         ra.subtract(bra);
86         ga.subtract(rga);
87         ga.subtract(gba);
88         ba.subtract(bra);
89         ba.subtract(gba);
90         //填充颜色区域
91         g2.setColor(new Color(red,0,0));
92         g2.fill(ra);
93         g2.setColor(new Color(0,green,0));
94         g2.fill(ga);
95         g2.setColor(new Color(0,0,blue));
96         g2.fill(ba);
97         g2.setColor(new Color(red,green,0));
98         g2.fill(rga);
99         g2.setColor(new Color(0,green,blue));
100        g2.fill(gba);
101        g2.setColor(new Color(red,0,blue));
102        g2.fill(bra);
103        g2.setColor(new Color(red,green,blue));
104        g2.fill(rgba);
105        //绘制三个圆
106        g2.setColor(Color.black);
107        g2.draw(rc);
108        g2.draw(gc);
109        g2.draw(bc);
110    }
111 }
```


ColorPanel 类使用red、green、blue三个整型字段表示基色分量。在paintComponent方法中(第67行),使用区域几何模型构造了三个相交的圆,这三个圆形成了七个区域,每个区域都使用基色的组合进行填充。例如,红色圆覆盖的区域使用红颜色填充,红色和蓝色圆覆盖的区域使用红色和蓝色分量的组合值进行填充,三个圆覆盖的中心区域使用三种颜色的组合值进行填充。三个圆绘制在黑色背景(的画布)上。

三个竖直的JSlider对象放在窗口的右边,它们用来控制ColorPanel中red、green和blue三个字段的值。滚动条的取值范围在0~255之间,当一个滚动条发生变化时,相应的ChangeListener将改变颜色的值,并重绘面板(第30~53行)。

3.2.3 涂色

setColor(Color c)方法属于旧的Graphics类,它仅使用纯色进行绘制。Java 2D的Graphics2D类使用一个更强大的setPaint(Paint p)方法来控制绘制颜色。Paint是一种颜色抽象,由Color类和其他一些类来实现Paint接口(见图3-2),与简单的纯色相比,它能够表示更多的属性。

GradientPaint类定义了一种颜色变化的涂色(Paint)方式。渐变涂色(gradient paint)由两个点和两种颜色定义。当位置从第一个点向第二个点移动时,颜色逐渐从第一种颜色变化到第二种颜色。渐变涂色可以是循环的,也可以是非循环的,循环的渐变涂色周期性地重复同样的模式。为了创建一种非循环的渐变涂色效果,使用如下的构造函数:

```
//构造非循环的渐变涂色
GradientPaint(float x1, float y1, Color c1, float x2, float y2, Color c2);
GradientPaint(Paint2D p1, Color c1, Paint2D p2, Color c2);
```

在这两个构造函数中,都给出了两个点和它们相关联的颜色。两点之间任意点的颜色,逐渐从一种指定的颜色变化到另一种颜色。循环的渐变涂色方式使用如下的构造函数创建:

```
//构造循环的渐变涂色
GradientPaint(float x1, float y1, Color c1, float x2, float y2, Color c2, boolean cycl);
GradientPaint(Paint2D p1, Color c1, Paint2D p2, Color c2, boolean cycl);
```

构造循环渐变涂色时,将最后一个参数设置为true。

TexturePaint类允许使用纹理模式(texture pattern)填充对象。一幅图像和锚点矩形用于定义一个纹理涂色方式。当绘制纹理的时候,图像重复地应用于平铺的矩形区域。一个TexturePaint对象使用如下的构造函数创建:

```
TexturePaint(BufferImage image, Rectangle2D anchor); //创建TexturePaint对象
```

其中,图像定义了绘制的纹理,锚点矩形说明了图像在用户空间中的位置。

程序清单3-2说明了使用渐变涂色和纹理涂色的图形绘制效果。图3-3是程序的运行结果。

程序清单3-2 TestPaints.java

```
1 package chapter3;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.image.*;
6 import javax.swing.*;
7 import java.awt.font.*;
```

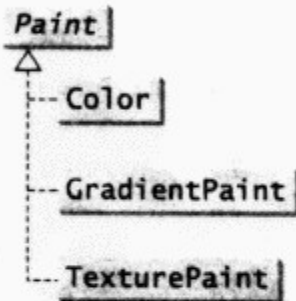


图3-2 Paint类的层次结构


```
8 import java.awt.geom.*;
9 import java.io.*;
10 import java.net.URL;
11 import javax.imageio.*;
12 //定义TestPaints类, 继承自JApplet类
13 public class TestPaints extends JApplet {
14     public static void main(String s[]) {
15         JFrame frame = new JFrame();           //创建主窗口
16         frame.setTitle("Gradient and Texture Paints");//设定窗口标题栏
17         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18         JApplet applet = new TestPaints();      //创建TestPaints对象并初始化
19         applet.init();
20         frame.getContentPane().add(applet);      //添加TestPaints对象到窗口对象中
21         frame.pack();
22         frame.setVisible(true);                 //设置窗口可见
23     }
24     //重写Applet初始化函数
25     public void init() {
26         JPanel panel = new JPanel();
27         getContentPane().add(panel);
28     }
29 }
30 //定义PaintPanel类, 继承自JPanel
31 class PaintPanel extends JPanel{
32     private BufferedImage image;
33
34     public PaintPanel() {
35         setPreferredSize(new Dimension(500, 500));
36         setBackground(Color.white);
37         URL url = getClass().getClassLoader().getResource
38             ("images/earth.jpg");           //获取图像URL
39         try {
40             image = ImageIO.read(url);      //读取图像
41         } catch (IOException ex) {
42             ex.printStackTrace();
43         }
44     }
45     //重写组件绘制函数
46     public void paintComponent(Graphics g) {
47         super.paintComponent(g);
48         Graphics2D g2 = (Graphics2D)g;     // 定义了一个循环的渐变涂色
49         GradientPaint gp = new GradientPaint(100,50,
50             Color.white, 150, 50, Color.gray, true);
51         g2.setPaint(gp);
52         g2.fillRect(100, 40, 300, 20);      //定义了一个纹理涂色
53         TexturePaint tp = new TexturePaint(image,
54             new Rectangle2D.Double(100, 100, image.getWidth(),
55                 image.getHeight()));
56         g2.setPaint(tp);
57         Shape ellipse = new Ellipse2D.Double(100, 100, 300, 200);
58         g2.fill(ellipse);
59         //定义了一个非循环的渐变涂色
60         GradientPaint paint = new GradientPaint(100,300, Color.white,
61             400, 400, Color.black);
62         g2.setPaint(paint);
63         Font font = new Font("Serif", Font.BOLD, 144);
```



```
63    g2.setFont(font);
64    g2.drawString("Java", 100, 400);
65 }
66 }
```



图3-3 使用循环渐变涂色、纹理涂色和非循环渐变涂色所绘制的形状

这个程序在`paintComponent`（第46行）方法中绘制了三个可视对象。在绘制的时候，使用了三种不同的涂色方式。最上方的矩形使用了循环渐变涂色，所使用的构造函数如下：

```
new GradientPaint(100, 50, Color.white, 150, 50, Color.gray, true);
```

渐变涂色定义了一个从白色到灰色的渐变过程，两个端点分别是（100, 50）和（150, 50）。这个模式是循环的，由于这两个点有相同的y坐标，因此可以看到一个垂直分布的图案。

椭圆使用图像定义的纹理涂色进行了填充。

```
new TexturePaint(image, new Rectangle2D.Double(100, 100, image.getWidth(),
image.getHeight()));
```

这个图像是一个`BufferedImage`对象，使用`ImageIO`类的`read`方法从图像文件中进行加载。文本字符串使用非循环的渐变涂色方式进行绘制：

```
new GradientPaint(100, 300, Color.white, 400, 400, Color.black);
```

渐变涂色方式的颜色从白色变到黑色，两个端点的位置分别是（100, 300）和（400, 400）。该涂色方式是非循环的，所用的颜色模式并不重复。

3.3 笔划

计算机图形学中所绘制的直线并不是一条宽度为0的理想直线。在实际的应用中，线必须有特定的形状，定义这些形状细节的属性称为笔划（stroke）。一个笔划可以包含线的宽度、虚线的风格、线帽的形状和线的连接风格等各种属性。

Java 2D使用`Stroke`接口表示笔划。类`BasicStroke`实现了`Stroke`接口。`BasicStroke`允许设置线的宽度、线帽、连接风格和虚线。`BasicStroke`的构造函数如下：

```
//BasicStroke()的各种构造函数
BasicStroke()
BasicStroke(float width)
BasicStroke(float width, int cap, int join)
BasicStroke(float width, int cap, int join, float miterlimit)
BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashphase)
```

66
67

参数width定义了画笔的宽度。参数cap定义了线帽的风格，可以取如下值：

```
CAP_BUTT
CAP_ROUND
CAP_SQUARE
```

参数join定义了连接风格，其取值如下：

```
JOIN_BEVEL
JOIN_MITER
JOIN_ROUND
```

参数miterlimit对JOIN_MITER进行限制，防止两条直线夹角很小的时候出现过长的连接。

dash数组通过设置ON/OFF段的长度定义了一个虚线模式。参数dashphase定义了虚线模式的起始点。

Graphics2D通过如下的方法设置当前的笔划：

```
void setStroke(Stroke s)
```

程序清单3-3演示了使用不同笔划设置对图形绘制效果的影响。程序中定义了三种线帽、三种连接风格和一个有两种虚线状态值的虚线数组。程序的运行结果如图3-4所示。

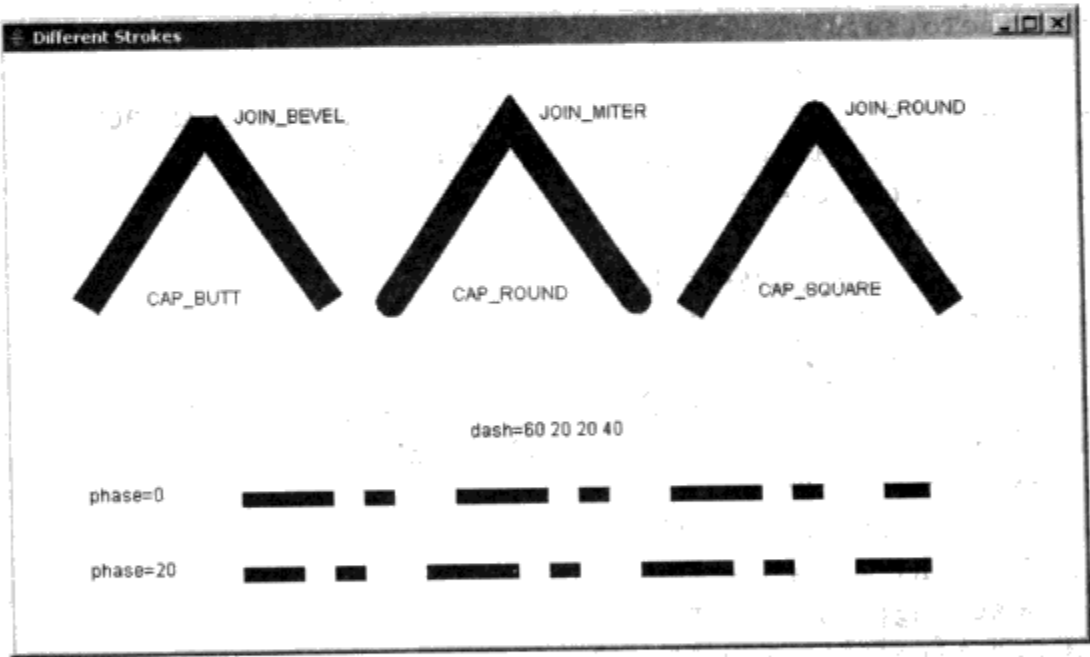


图3-4 使用不同线帽、连接风格和虚线的笔划示例

程序清单3-3 TestStrokes.java

```
1 package chapter3;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.geom.*;
7 //定义TestStrokes类，继承自JApplet类，演示不同类型的笔划绘制
8 public class TestStrokes extends JApplet {
```



```

9  public static void main(String s[]) {
10     JFrame frame = new JFrame();//创建主窗口
11     frame.setTitle("Different Strokes");//设置窗口标题栏
12     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13     JApplet applet = new TestStrokes();//创建TestStrokes对象
14     applet.init();
15     frame.getContentPane().add(applet); //将Applet添加到主窗口
16     frame.pack();
17     frame.setVisible(true);//设置窗口可见
18 }
19 //重写Applet初始化函数
20 public void init() {
21     JPanel panel = new StrokePanel();
22     getContentPane().add(panel);
23 }
24 }
25 //定义StrokePanel类, 继承自JPanel类, 负责绘制
26 class StrokePanel extends JPanel {
27     public StrokePanel() {
28         setPreferredSize(new Dimension(700, 400));//设置组件首选大小
29         setBackground(Color.white);//设置组件背景颜色为白色
30     }
31     //重写组件绘制函数
32     public void paintComponent(Graphics g) {
33         super.paintComponent(g);
34         Graphics2D g2 = (Graphics2D)g;
35         GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
36         path.moveTo(0,120); //定义了一个两条线段相连的路径
37         path.lineTo(80,0);
38         path.lineTo(160,120);
39         Stroke stroke = new BasicStroke(20, BasicStroke.CAP_BUTT,
40             BasicStroke.JOIN_BEVEL); //构造一个笔划对象
41         g2.setStroke(stroke);
42         g2.translate(50,50);//平移
43         g2.draw(path);//绘制路径和字符串
44         g2.drawString("JOIN_BEVEL",100,0);
45         g2.drawString("CAP_BUTT", 40, 120);
46         stroke = new BasicStroke(20, BasicStroke.CAP_ROUND,
47             BasicStroke.JOIN_MITER); //构造笔划对象
48         g2.setStroke(stroke);
49         g2.translate(200,0);//平移
50         g2.draw(path);//绘制路径和字符串
51         g2.drawString("JOIN_MITER",100,0);
52         g2.drawString("CAP_ROUND", 40, 120);
53         stroke = new BasicStroke(20, BasicStroke.CAP_SQUARE,
54             BasicStroke.JOIN_ROUND); //构造笔划对象
55         g2.setStroke(stroke);
56         g2.translate(200,0);//平移
57         g2.draw(path);//绘制路径和字符串
58         g2.drawString("JOIN_ROUND",100,0);
59         g2.drawString("CAP_SQUARE", 40, 120);
60         float[] dashArray = {60,20,20,40};
61         float dashPhase = 0;//构造一个虚线的笔划对象
62         stroke = new BasicStroke(10, BasicStroke.CAP_BUTT,
63             BasicStroke.JOIN_BEVEL, 0, dashArray, dashPhase);
64         g2.setStroke(stroke);

```

```
65     g2.translate(-400,200); //平移
66     g2.drawLine(100, 50, 550, 50); //绘制线段和字符串
67     g2.drawString("dash=60 20 20 40", 250, 10);
68     g2.drawString("phase=0", 0, 50);
69     dashPhase = 20;
70     stroke = new BasicStroke(10, BasicStroke.CAP_BUTT,
71         BasicStroke.JOIN_BEVEL, 0, dashArray, dashPhase); //构造一个虚线笔划对象
72     g2.setStroke(stroke);
73     g2.translate(0,50); //平移
74     g2.drawLine(100, 50, 550, 50); //绘制线段和字符串
75     g2.drawString("phase=20", 0, 50);
76 }
77 }
```

GeneralPath对象由两个相连的线段构造而成（第34~37行）。在窗口的最上面一行将该路径绘制了三次。在绘制过程中，使用了具有不同线帽和连接风格的BasicStroke实例。为了显示笔划样式的细节，把笔划的宽度设为20。

窗口最下面两行显示的是虚线。虚线数组定义为{60, 20, 20, 40}（第59行），两种笔划的线宽都为10，但是相应的虚线值分别为0和20。从显示的结果可以清楚地看出非零状态虚线模式的变化。

69

3.4 仿射变换

几何对象在绘制以前，需要经过一系列变换。在计算机图形学里一般使用的一类几何变换，称作仿射变换（affine transform）。仿射变换保留线的平行性质。维持任意两点距离不变的仿射变换，也称为等距变换（isometry）、欧几里得运动（Euclidean motion）或刚体运动（rigid motion）。常见的仿射变换包括：

- 平移
- 旋转
- 反射
- 缩放
- 错切

在平移（translation）变换中，对象的所有点都要移动一个固定的值（如图3-5所示）。它由x方向和y方向的移动量所决定。平移是等距变换，它并不改变对象的长度和角度。图3-5给出了一个平移量为（3，-1）的平移变换。图中的对象向右平移了3个单位，向上平移了1个单位。

在旋转（rotation）变换中，对象绕某个点旋转一个角度（如图3-6所示）。旋转变换由旋转点和旋转角度决定。尽管旋转变换改变了形状的方向，但它也是一种等距变换。图3-6显示了一个绕原点旋转45度角的旋转变换。

反射（reflection）变换沿着某一条直线建立该对象的镜像（如图3-7所示）。反射变换的结果由变换直线决定。虽然反射变换改变了角度的方向，但它也是一种等距变换。图3-7显示了一条沿着介于x轴和y轴之间呈45度角的直线所进行的反射变换。

缩放（scaling）变换以一定的比例，在x和y方向上改变对象的大小（如图3-8所示）。缩放变换不是等距变换，因为它改变了被变换对象的距离和角度。不过，这种变换仍然维持了线条的平行性质。图3-8显示了一个缩放因子为（1.5, 2）的缩放变换。

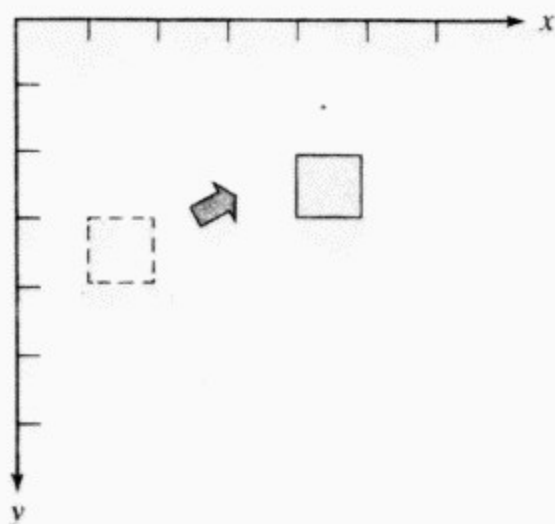


图3-5 平移量为 (3, -1) 的平移变换

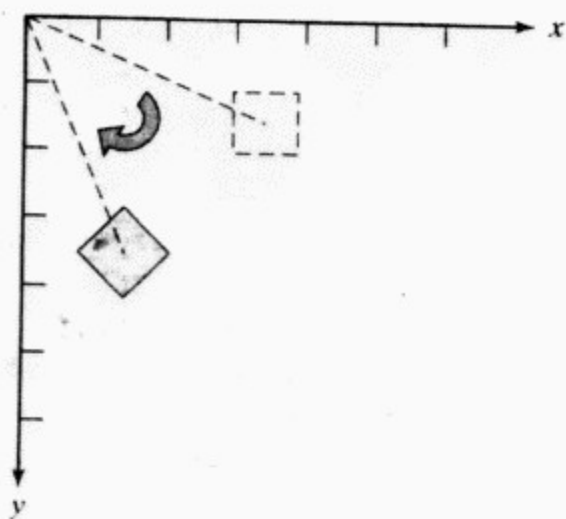


图3-6 绕原点旋转

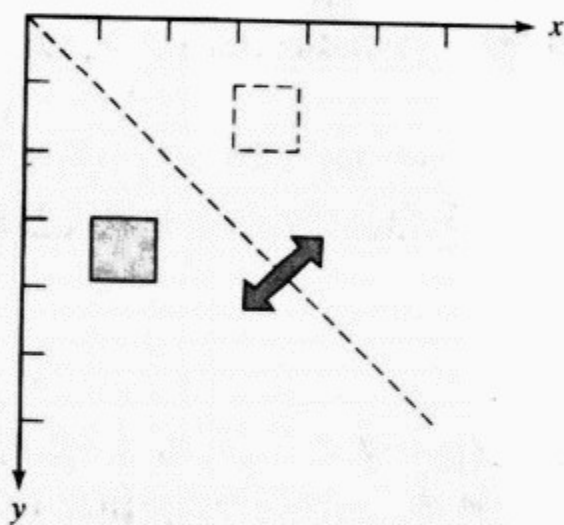


图3-7 关于对角线的反射

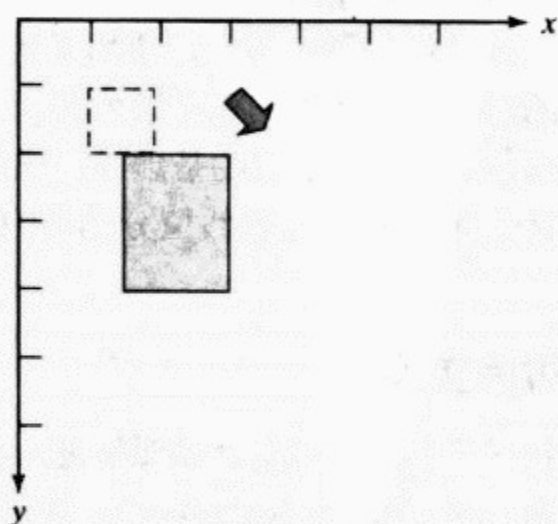


图3-8 因子为 (1.5, 2) 的缩放变换

70
71

关于某条直线的错切 (shearing) 变换, 是将变换对象上的点移动一定的距离, 该距离与点到变换直线的有符号距离成正比 (如图3-9所示)。点的移动平行于这条直线, 在直线上的点并不移动, 在直线反方向上的点以相反的方向移动。错切变换不是等距变换, 但是它仍维持线条的平行性质。图3-9显示了一个沿水平直线 $y=2$ 进行因子为1.0的错切变换。

在数学上, 一个二维仿射变换可以用一个 3×3 的矩阵 (matrix) 表示。仿射变换使用 3×3 的矩阵而不使用 2×2 的矩阵, 这是因为在二维空间中, 诸如平移变换之类的变换并不是线性的。利用齐次坐标的概念, 把点坐标的向量表示增加一个维, 就可以在线性框架下来处理仿射变换了。在附录A中, 更详细地讨论了有关矩阵和齐次坐标的知识。

对于基本的变换, 很容易直接寻找它们的仿射矩阵。绕原点的旋转角度为 θ 的旋转变换, 其变换矩阵可以表示为:

$$\begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

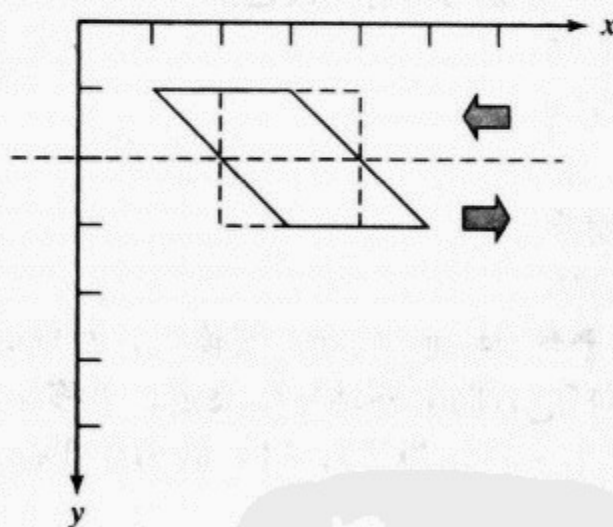


图3-9 沿水平虚线进行的因子为1的错切变换

平移量为 (a, b) 的平移变换矩阵可以表示为：

$$\begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix}$$

缩放因子为 (α, β) 的缩放变换矩阵可以表示为：

$$\begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

一个关于直线 $y = kx$ 的反射变换矩阵可以表示为：

$$\begin{bmatrix} \frac{2}{1+k^2} - 1 & \frac{2k}{1+k^2} & 0 \\ \frac{2k}{1+k^2} & \frac{2k^2}{1+k^2} - 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

72

关于 x 轴、因子为 s 的错切变换矩阵可以表示为：

$$\begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Java 2D 使用 `AffineTransform` 类定义仿射变换，它使用最简便的方法建立前面定义的大部分基本仿射变换。`AffineTransform` 类的下列方法用于定义这些变换：

```
void setToIdentity()      //单位变换
void setToRotation(double theta) //沿原点的旋转变换
void setToRotation(double theta, double x, double y) //沿某个点(x, y)的旋转变换
void setToScale(double sx, double sy) //缩放变换
void setToShear(double shx, double shy) //错切变换
void setToTranslation(double tx, double ty) //平移变换
```

这里没有定义反射变换的方法。然而，可以通过仿射矩阵来定义反射变换，如下矩阵是关于 y 轴的反射变换矩阵：

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`AffineTransform` 类的构造函数和方法，可以直接设置变换矩阵的前两行：

```
//仿射变换的构造函数
AffineTransform(double m00, double m10, double m01, double m11, double m02,
double m12)
AffineTransform(float m00, float m10, float m01, float m11, float m02, float m12)
AffineTransform(double[] flatmatrix)
AffineTransform(float[] flatmatrix)
//仿射变换的参数设置函数
void setTransform(double m00, double m10, double m01, double m11, double m02,
double m12)
```

由于仿射矩阵的最后一行总是 $(0 \ 0 \ 1)$ ，因此在参数表中将它略去了。反射变换可以定义

成如下形式：

```
transform.setTransform(-1, 0, 0, 1, 0, 0);
```

由于AffineTransform类允许使用负因子的缩放变换，因此反射变换也可以定义成特殊的缩放变换：

```
transform.setToScale(-1, 1);
```

AffineTransform对象既可以用于对象变换（object transformation），也可以用于视图变换（viewing transformation）。下面的AffineTransform类方法用于几何对象的变换：

//用于对象变换的函数

```
Shape createTransformedShape(Shape shape)
void transform(double[] src, int srcOff, double[] dst, int dstOff, int numPts)
void transform(double[] src, int srcOff, float[] dst, int dstOff, int numPts)
void transform(float[] src, int srcOff, double[] dst, int dstOff, int numPts)
void transform(float[] src, int srcOff, float[] dst, int dstOff, int numPts)
Point2D transform(Point2D src, Point2D dst)
Point2D transform(Point2D[] src, int srcOff, Point2D[] dst, int dstOff, int numPts)
void detlaTransform(double[] src, int srcOff, double[] dst, int dstOff, int numPts)
Point2D detlaTransform(Point2D src, Point2D dst)
```

createTransformedShape方法对整个形状进行变换，transform方法对一系列点进行变换，detlaTransform方法对一系列向量进行变换。

视图变换可以通过Graphics2D对象中的变换实现。Graphics2D类有如下的方法操纵自己的变换：

//用于观察变换的函数

```
void setTransform(AffineTransform tx)
void transform(AffineTransform tx)
```

setTransform方法使用给定的AffineTransform对象代替当前的变换。transform方法使用右边的AffineTransform对象连接当前的变换。

程序清单3-4通过交互方式给出了仿射变换的效果。用户可以使用鼠标对一个图形对象进行变换。通过菜单选择平移、旋转、缩放、错切、反射变换。图3-10显示了程序的运行结果。

程序清单3-4 Transformations.java

```
1 package chapter3;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.event.*;
6 import java.util.*;
7 import javax.swing.*;
8 //定义Transformations类，继承自JApplet类，实现了ActionListener接口
9 public class Transformations extends JApplet implements //多种仿射变换
10     ActionListener {
11     public static void main(String s[]) {
12         JFrame frame = new JFrame();//生成主窗口
13         frame.setTitle("Affine Transforms");//设置窗口标题栏
14         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15         JApplet applet = new Transformations();//创建Transformations对象
16         applet.init();//初始化
17         frame.getContentPane().add(applet);//将Transformations对象添加到主窗口
18         frame.pack();
```

```
19     frame.setVisible(true); //设置窗口可见
20 }
21
22 TransformPanel panel = null; //声明变换面板类TransformPanel对象
23 //重写JApplet初始化方法
24 public void init() {
25     JMenuBar mb = new JMenuBar(); //生成菜单栏及相应的菜单选项
26     setJMenuBar(mb);
27     JMenu menu = new JMenu("Transforms");
28     mb.add(menu);
29     JMenuItem mi = new JMenuItem("Translation");
30     mi.addActionListener(this);
31     menu.add(mi);
32     mi = new JMenuItem("Rotation");
33     mi.addActionListener(this);
34     menu.add(mi);
35     mi = new JMenuItem("Scaling");
36     mi.addActionListener(this);
37     menu.add(mi);
38     mi = new JMenuItem("Shearing");
39     mi.addActionListener(this);
40     menu.add(mi);
41     mi = new JMenuItem("Reflection");
42     mi.addActionListener(this);
43     menu.add(mi);
44
45     panel = new TransformPanel(); //初始化变换面板并加入JApplet内容窗格
46     getContentPane().add(panel);
47 }
48 //事件响应函数，根据用户选择设置变换类型
49 public void actionPerformed(ActionEvent ev) {
50     String command = ev.getActionCommand();
51     if ("Translation".equals(command)) {
52         panel.transformType = panel.TRANSLATION;
53     } else if ("Rotation".equals(command)) {
54         panel.transformType = panel.ROTATION;
55     } else if ("Scaling".equals(command)) {
56         panel.transformType = panel.SCALING;
57     } else if ("Shearing".equals(command)) {
58         panel.transformType = panel.SHEARING;
59     } else if ("Reflection".equals(command)) {
60         panel.transformType = panel.REFLECTION;
61     }
62 }
63 }
64 //定义TransformPanel类，继承自JPanel类，实现鼠标事件侦听器
65 class TransformPanel extends JPanel
66     implements MouseListener, MouseMotionListener {
67     static final int NONE = 0; //变换类型常量
68     static final int TRANSLATION = 1;
69     static final int ROTATION = 2;
70     static final int SCALING = 3;
71     static final int SHEARING = 4;
72     static final int REFLECTION = 5;
73
74     int transformType = NONE; //保存当前变换类型，默认为无变换
```



```

75 Shape drawShape = null;
76 Shape tempShape = null;
77 Point p = null;
78 int x0 = 400;
79 int y0 = 300;
80
81 public TransformPanel() {
82     super();
83     setPreferredSize(new Dimension(800, 600)); // 设定组件首选大小
84     setBackground(Color.white); // 设定背景颜色为白色
85     drawShape = new Rectangle(-50, -50, 100, 100); // 定义矩形形状
86     addMouseListener(this); // 添加事件侦听器
87     addMouseMotionListener(this);
88 }
89 // 重写组件绘制方法
90 public void paintComponent(Graphics g) {
91     super.paintComponent(g);
92     Graphics2D g2 = (Graphics2D)g;
93     g2.translate(x0, y0); // 平移
94     g2.drawLine(-200, 0, 200, 0); // 绘制坐标轴
95     g2.drawLine(0, -200, 0, 200);
96     g2.draw(drawShape); // 绘制待变换对象
97 }
98
99 public void mouseClicked(MouseEvent ev) {
100 }
101
102 public void mouseEntered(MouseEvent ev) {
103 }
104
105 public void mouseExited(MouseEvent ev) {
106 }
107 // 处理鼠标按下事件
108 public void mousePressed(MouseEvent ev) {
109     p = ev.getPoint(); // 保存当前点
110 }
111 // 处理鼠标释放事件，进行仿射变换：平移、旋转、缩放、错切和反射
112 public void mouseReleased(MouseEvent ev) {
113     Graphics g = getGraphics();
114     Point p1 = ev.getPoint();
115     AffineTransform tr = new AffineTransform();
116     switch (transformType) { // 根据不同变换类型分别进行处理
117         case TRANSLATION:
118             tr.setToTranslation(p1.x-p.x, p1.y-p.y);
119             break;
120         case ROTATION:
121             double a = Math.atan2(p1.y-y0, p1.x-x0) - Math.atan2
122                 (p.y-y0, p.x-x0);
123             tr.setToRotation(a);
124             break;
125         case SCALING:
126             double sx = Math.abs((double)(p1.x-x0)/(p.x-x0));
127             double sy = Math.abs((double)(p1.y-y0)/(p.y-y0));
128             tr.setToScale(sx, sy);
129             break;
130         case SHEARING:

```

```

131         double shx = ((double)(p1.x-x0)/(p.x-x0))-1;
132         double shy = ((double)(p1.y-y0)/(p.y-y0))-1;
133         tr.setToShear(shx, shy);
134         break;
135     case REFLECTION:
136         tr.setTransform(-1,0,0,1,0,0);
137         break;
138     }
139     drawShape = tr.createTransformedShape(drawShape); //进行变换
140     repaint(); //重绘
141 }
142
143 public void mouseMoved(MouseEvent ev) {
144 }
145 //处理鼠标拖动事件，与mouseReleased函数类似，进行仿射变换
146 public void mouseDragged(MouseEvent ev) {
147     Point p1 = ev.getPoint();
148     AffineTransform tr = new AffineTransform();
149     switch (transformType) {
150         case TRANSLATION:
151             tr.setToTranslation(p1.x-p.x, p1.y-p.y);
152             break;
153         case ROTATION:
154             double a = Math.atan2(p1.y-y0, p1.x-x0) - Math.atan2
155                 (p.y-y0, p.x-x0);
156             tr.setToRotation(a);
157             break;
158         case SCALING:
159             double sx = Math.abs((double)(p1.x-x0)/(p.x-x0));
160             double sy = Math.abs((double)(p1.y-y0)/(p.y-y0));
161             tr.setToScale(sx, sy);
162             break;
163         case SHEARING:
164             double shx = ((double)(p1.x-x0)/(p.x-x0))-1;
165             double shy = ((double)(p1.y-y0)/(p.y-y0))-1;
166             tr.setToShear(shx, shy);
167             break;
168         case REFLECTION:
169             tr.setTransform(-1,0,0,1,0,0);
170             break;
171     }
172     Graphics2D g = (Graphics2D) getGraphics();
173     g.setXORMode(Color.white);
174     g.translate(x0, y0); //绘制变换前的形状
175     if (tempShape != null)
176         g.draw(tempShape); //对整个形状进行仿射变换
177     tempShape = tr.createTransformedShape(drawShape); //绘制变换后的形状
178     g.draw(tempShape);
179 }
180 }

```

76

这个程序起初显示了坐标系统中的一组坐标轴和一个矩形，该程序还定义了一个菜单，从菜单列表中选择平移、旋转、缩放、错切和反射（第25~43行）。为了进行特定的变换，在想要变换的方向上拖动鼠标，鼠标移动的量用来决定针对该对象要施加的变换幅度。

在这个程序中，定义了Transformations类和TransfromPanel类这两个类。Transformations类

是JApplet类的子类，它包含选择当前仿射变换的菜单。这个类的事件处理程序负责处理菜单选择动作，并将TransformPanel类的transformType变量设置为合适的值。

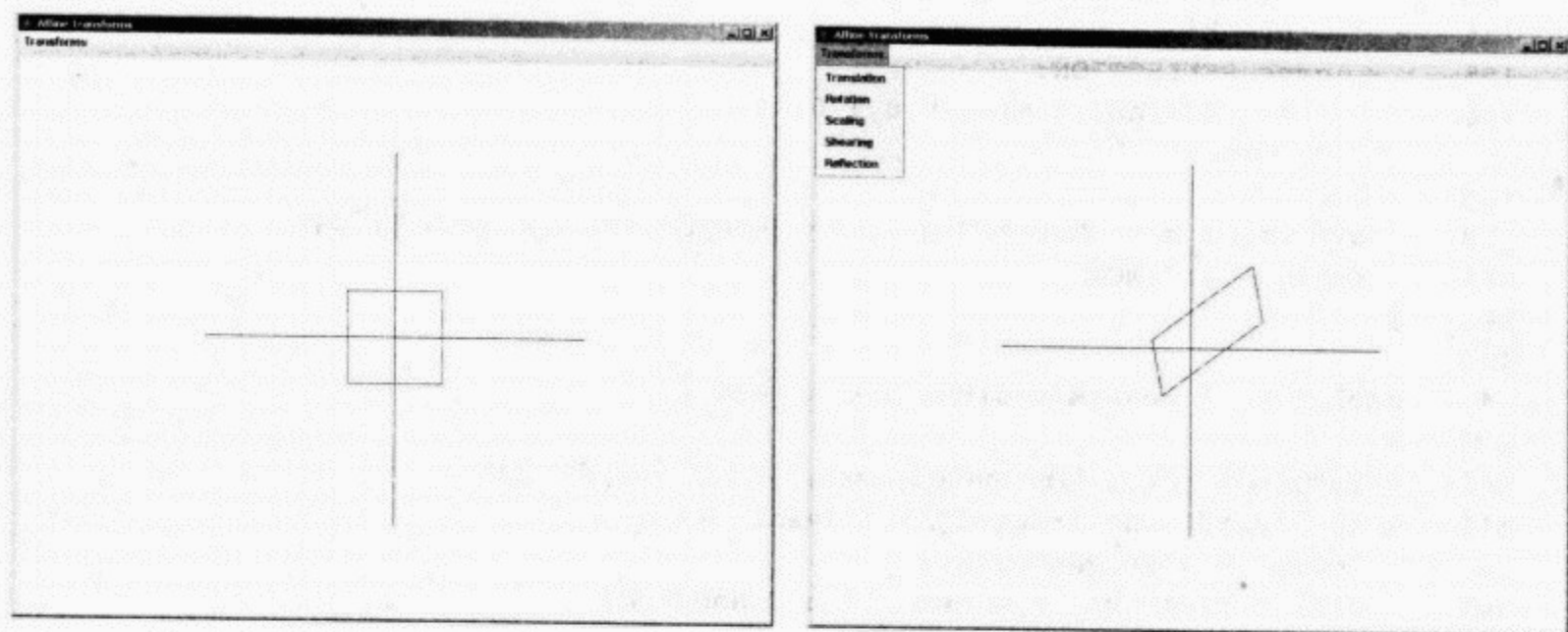


图3-10 应用到一个矩形的仿射变换

TransformPanel类是JPanel类的子类，它处理鼠标事件。当需要一个变换时，根据变换类型和鼠标移动情况，设定一个AffineTransform对象。据此，使用createTransformedShape方法将该变换作为目标变换，应用到待变换的形状之上（第139行）。

初始的形状是一个矩形，但是，逐步累积的变换效果，使得变换后的形状变成了当前的样子。变换并不影响诸如坐标轴之类的其他对象，因为这些变换只是针对形状局部进行的一种对象变换。

在用户拖放鼠标的时候，用到了橡皮条技术来提供视觉线索，这个方法与程序清单2-3中的相同。XOR绘制模式用于绘制和删除当前形状。

3.5 复合变换

变换能够组合起来形成新的变换。例如，可以先使用旋转变换，然后使用平移变换。任何仿射变换的组合形式（composition）仍然是仿射变换，任何刚体变换的组合形式仍然是刚体变换。相反，一个变换可以分解成一系列（通常是更简单的）变换。

复合变换的变换矩阵是单个变换矩阵的乘积。例如，假设 M_1 、 M_2 、 M_3 分别是仿射变换 T_1 、 T_2 、 T_3 的变换矩阵，相应地，复合变换 $T_1 \circ T_2 \circ T_3$ 的仿射变换矩阵是 $M_1 M_2 M_3$ 。注意复合变换的运算顺序是不可交换的（noncommutative），因此变换的复合顺序很重要。在本书的记法中，复合变换是从右至左结合的。例如，对一个点 p 应用复合变换 $T_1 \circ T_2 \circ T_3$ 时，变换的顺序是 T_3 、 T_2 、 T_1 ：

$$(T_1 \circ T_2 \circ T_3)(p) = T_1(T_2(T_3(p)))$$

由简单变换构造复杂变换时，复合变换是很有用的。假如要将某对象绕点 $(3, 4)$ 旋转30度，首先需要把点 $(3, 4)$ 移动至原点，然后再进行一个绕原点的30度的旋转变换，最后再把原点平移回点 $(3, 4)$ 。使用这三个变换的复合形式，同样可以得到所需要的变换结果。把点 $(3, 4)$ 平移至原点的平移变换矩阵如下：

$$\begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix}$$

绕原点30度的旋转变换矩阵如下：

$$\begin{bmatrix} \sqrt{3}/2 & -1/2 & 0 \\ 1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

第二个平移变换矩阵如下：

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

对这些变换进行复合，最终的变换矩阵可以表示为：

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \sqrt{3}/2 & -1/2 & 0 \\ 1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix}$$

在Java 2D中，AffineTransform类提供如下的方法支持复合变换：

//相应的复合变换函数

```
void rotate(double theta)
void rotate(double theta, double x, double y)
void scale(double sx, double sy)
void shear(double shx, double shy)
void translate(double tx, double ty)
```

78

和前一节介绍的setTo*方法不同，这些方法并不清除当前对象已经用到的变换，而是对当前的变换和新近指定的变换进行复合，新的变换添加至当前变换的右边。除了上面提到的简单变换之外，也可以将当前的变换和另一个仿射变换对象复合起来：

```
//直接将当前变换和另外一个仿射变换对象复合
void concatenate(AffineTransform tx)
void preConcatenate(AffineTransform tx)
```

第一个方法在当前变换的右边连接其他变换，第二个方法在当前变换的左边连接其他变换。

需要注意到，变换的复合顺序是从左至右，而上面的方法（除preConcatenate以外）是从右至左连接变换的。如果调用上面的方法创建一个复合变换，这些变换将以相反的顺序进行调用。例如，考虑如下的代码：

```
AffineTransform transform = new AffineTransform();
transform.rotate(Math.PI/3);
transform.scale(2,0.3);
transform.translate(100,200);
```

运用到的第一个变换是平移变换，最后一个变换是旋转变换。

程序清单3-5演示了复合变换的使用方法。该程序将一椭圆沿其中心进行旋转，并且该椭圆的中心不在原点。程序首先将椭圆中心平移至原点，然后绕原点进行旋转，最后再把椭圆中心平移回最初的位置（如图3-11所示）。

程序清单3-5 Composition.java

```
1 package chapter3;
```



```
2
3 import javax.swing.*;
4 import java.awt.*;
5 import java.awt.geom.*;
6 //定义Composition类, 继承自JApplet类, 演示变换组合
7 public class Composition extends JApplet {
8     public static void main(String s[]) {
9         JFrame frame = new JFrame();//生成主窗口
10        frame.setTitle("Transformation Composition");//设置窗口标题栏
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12        JApplet applet = new Composition();//创建Composition对象
13        applet.init();//初始化
14        frame.getContentPane().add(applet); //添加到主窗口
15        frame.pack();
16        frame.setVisible(true);//设置窗口可见
17    }
18    //重写JApplet初始化函数
19    public void init() {
20        JPanel panel = new CompositionPanel();//创建CompositionPanel对象
21        getContentPane().add(panel);//加入JApplet内容窗格
22    }
23 }
24 //定义CompositionPanel类, 继承自JPanel类
25 class CompositionPanel extends JPanel {
26     public CompositionPanel() {
27         setPreferredSize(new Dimension(640, 480));//设置组件首选大小
28         this.setBackground(Color.white);//设置背景为白色
29     }
30     //重写组件绘制函数
31     public void paintComponent(Graphics g) {
32         super.paintComponent(g);
33         Graphics2D g2 = (Graphics2D)g;
34         g2.translate(100,100);//平移
35         Shape e = new Ellipse2D.Double(300, 200, 200, 100); //绘制一个椭圆
36         g2.setColor(new Color(160,160,160));
37         g2.fill(e);//填充椭圆
38         AffineTransform transform = new AffineTransform();
39         transform.translate(-400,-250); //平移
40         e = transform.createTransformedShape(e);//生成平移后图形
41         g2.setColor(new Color(220,220,220));//设置颜色
42         g2.fill(e);//填充椭圆
43         g2.setColor(Color.black);
44         g2.drawLine(0, 0, 150, 0);//绘制两条线段
45         g2.drawLine(0, 0, 0, 150);
46         transform.setToRotation(Math.PI / 6.0);//旋转
47         e = transform.createTransformedShape(e);//生成旋转后图形
48         g2.setColor(new Color(100,100,100));//设置颜色
49         g2.draw(e); //绘制椭圆
50         transform.setToTranslation(400, 250); //平移
51         e = transform.createTransformedShape(e); //生成平移后图形
52         g2.setColor(new Color(0,0,0));//设置颜色
53         g2.draw(e);//绘制椭圆
54     }
55 }
```

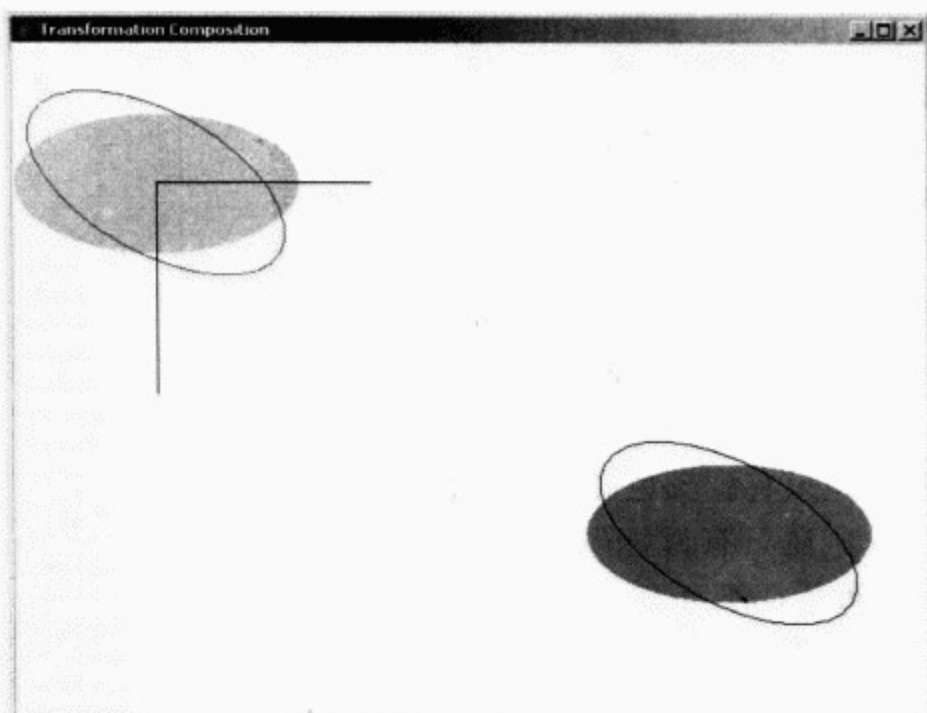



图3-11 一个复合变换，分别进行了平移变换、旋转变换和另一个平移变换

这个程序给出了在复合变换不同阶段对象的形状。椭圆的形状最初由边界矩形 (300, 200, 200, 100) 构造，因而矩形的中心位于 (400, 250) (第35行)。变换的目标是将椭圆绕点 (400, 250) 旋转30度。为了完成这个变换，需要使用三个变换的复合。首先平移 (-400, -250) 使椭圆中心位于原点，然后绕原点旋转30°，旋转之后椭圆中心依旧在原点，最后平移 (400, 250)，使椭圆中心再次回到最初的 (400, 250) 位置。这三个变换的复合结果就是所需的变换结果。

为了显示图形的所有部分，可使用视图变换把世界坐标系从左上角移至点 (100, 100) (第34行)，在新的原点处画出x轴和y轴。椭圆经过每个阶段的变换之后，使用不同的灰度显示出来。程序并没有对旋转后的椭圆进行填充。

3.6 透明度和合成规则

合成规则 (compositing rule) 决定了绘制重叠对象的结果。通过选择合成规则，可以获得诸如不同的透明度之类的各种视觉效果。

要建立合成规则，需要用到 α 通道 (α -channel) 的概念。 α 通道可以视作颜色属性的一部分，用它来描述透明度。一个 α 值是介于0.0和1.0之间的数值，0.0表示完全透明，1.0表示完全不透明。

给定源与目标像素颜色和 α 值，Porter-Duff 规则 (Porter-Duff rule) 用源和目标像素值的线性组合形式，定义形成的颜色和 α 值：

$$\begin{aligned}\alpha \cdot C &= F_s \cdot \alpha_s \cdot C_s + F_d \cdot \alpha_d \cdot C_d \\ \alpha &= F_s \cdot \alpha_s + F_d \cdot \alpha_d\end{aligned}$$

通常，将颜色分量预先与它们的 α 值求积，可以提高运算的速度。等式中两个系数 F_s 和 F_d 选取不同的值，就定义了不同的复合规则。目前总共有12种Porter-Duff规则可用，它们所用的系数列在表3-1中。

Porter-Duff规则可以由概率模型 (probabilistic model) 系统地推导出来。颜色的 α 值可以解释为该颜色的显示概率，或者更具体地说，它是该颜色覆盖的像素区域比例值。按照源与目标像素各自对应的 α 值对两个像素的颜色值进行合成时，需要考虑的四种情况是：只使用源像素颜色，只使用目标像素颜色，同时使用源与目标两个像素的颜色，以及不使用源与目标任何像素的颜色等。图3-12演示了这四种情况，它们发生的概率分别为 $\alpha_s(1-\alpha_d)$ 、 $\alpha_d(1-\alpha_s)$ 、 $\alpha_s\alpha_d$ 和

$(1-\alpha_s)(1-\alpha_d)$ 。合成规则决定当一种颜色出现时是否保留这种颜色，在仅有源像素颜色的情况下，合成规则选择保留源像素颜色或忽略它。在仅有目标像素颜色的情况下，可以选择目标像素颜色，也可以忽略目标像素颜色。当两种像素颜色都存在时，合成规则既可以选择源像素颜色，也可以选择目标像素颜色，或者两种像素颜色都不选。当两种像素颜色都不存在时，合成规则不选择任何像素颜色。因此，这个模型总的规则个数是

81

例如，SrcOver规则在只有源颜色和两种颜色都存在的情况下，都选择源颜色；在只有目标颜色存在的情况下，选择目标颜色；在两种颜色都不存在的情况下，什么都不选。因此源颜色在合成中出现的概率为 $\alpha_s(1-\alpha_d) + \alpha_s\alpha_d = \alpha_s$ ，目标颜色出现的概率为 $\alpha_d(1-\alpha_s)$ 。据此可以确定，系数 $F_s = 1$ ， $F_d = (1-\alpha_s)$ ，如表3-1所示。

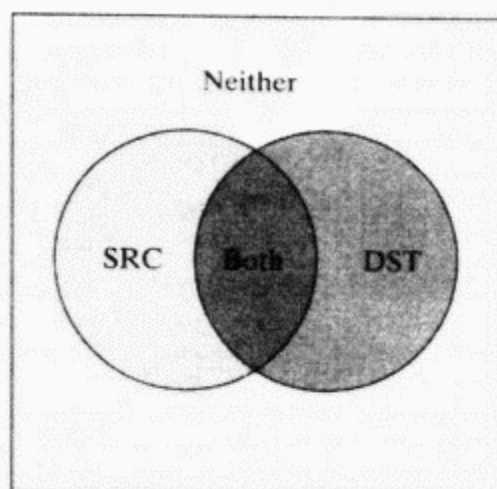


图3-12 合成概率模型中不同颜色出现的四种情况

表 3-1

Porter-Duff 规则	F_s	F_d
Clear	0	0
SrcOver	1	$1-\alpha_s$
DstOver	$1-\alpha_d$	1
SrcIn	α_d	0
DstIn	0	α_s
SrcOut	$1-\alpha_d$	0
DstOut	0	$1-\alpha_s$
Src	1	0
Dst	0	1
SrcAtop	α_d	$1-\alpha_s$
DstAtop	$1-\alpha_d$	α_s
Xor	$1-\alpha_d$	$1-\alpha_s$

早期版本的Java 2D支持表3-1中的前8个规则。从J2SDK 1.4起，支持全部的12个规则。AlphaComposite类封装了这些规则，AlphaComposite的规则实例对象可以通过该类的静态字段获取，静态字段的名字在表3-1中给出。对一个Graphics2D对象应用合成规则，可以简单地调用setComposite方法。例如，下面的语句将合成规则设置为SrcIn：

```
Graphics2D g2 = (Graphics2D)g;
g2.setComposite(AlphaComposite.SrcIn);
```

程序清单3-6举例说明了如何使用AlphaComposite类来实现Porter-Duff规则。在这个例子中，使用了12种Porter-Duff规则对可视化对象进行了绘制。本程序可以通过在显示面板上点击鼠标选择不同的规则。程序运行效果如图3-13所示。

程序清单3-6 Compositiong.java

```
1 package chapter3;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.image.*;
```



```
6 import javax.swing.*;
7 import java.awt.font.*;
8 import java.awt.geom.*;
9 import java.io.*;
10 import java.net.URL;
11 import javax.imageio.*;
12 //定义Compositing类, 继承自JApplet类
13 public class Compositing extends JApplet {
14     public static void main(String s[]) {
15         JFrame frame = new JFrame();//生成主窗口
16         frame.setTitle("Compositing Rules");//设置窗口标题栏
17         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设定关闭处理
18         JApplet applet = new Compositing(); //生成Compositing对象
19         applet.init();//初始化
20         frame.getContentPane().add(applet); //添加applet到主窗口
21         frame.pack();
22         frame.setVisible(true);//设置窗口可见
23     }
24     //重写JApplet初始化方法
25     public void init() {
26         JPanel panel = new CompositPanel();//创建CompositPanel对象
27         getContentPane().add(panel);//加入内容窗格
28     }
29 }
30 //定义CompositPanel类, 继承自JPanel类, 实现MouseListener接口
31 class CompositPanel extends JPanel implements MouseListener {
32     BufferedImage image;
33     int[] rules = {AlphaComposite.CLEAR, AlphaComposite.SRC_OVER,
34         AlphaComposite.DST_OVER, AlphaComposite.SRC_IN,
35         AlphaComposite.DST_IN, AlphaComposite.SRC_OUT,
36         AlphaComposite.DST_OUT, AlphaComposite.SRC,
37         AlphaComposite.DST, AlphaComposite.SRC_ATOP,
38         AlphaComposite.DST_ATOP, AlphaComposite.XOR};//定义合成规则数组
39     int ruleIndex = 0;
40
41     public CompositPanel() {
42         setPreferredSize(new Dimension(500, 400));//设置组件首选大小
43         setBackground(Color.white);//设置背景颜色为白色
44         URL url =
45             getClass().getClassLoader().getResource("images/earth.jpg");
46         try {
47             image = ImageIO.read(url);//读取图像
48         } catch (IOException ex) {
49             ex.printStackTrace();
50         }
51         addMouseListener(this);//添加鼠标事件侦听器
52     }
53     //重写组件绘制方法
54     public void paintComponent(Graphics g) {
55         super.paintComponent(g);
56         Graphics2D g2 = (Graphics2D)g;
57         g2.drawImage(image, 100, 100, this);//绘制图像
58         AlphaComposite ac = //生成合成规则实例
59             AlphaComposite.getInstance(rules[ruleIndex], 0.4f);
60         g2.setComposite(ac); //设置合成规则
61         Shape ellipse = new Ellipse2D.Double(50, 50, 120, 120);//创建椭圆
```



```

62     g2.setColor(Color.red); // 设置为红色
63     g2.fill(ellipse); // 进行填充
64     g2.setColor(Color.orange); // 设置为橙色
65     Font font = new Font("Serif", Font.BOLD, 144); // 创建设置字体
66     g2.setFont(font);
67     g2.drawString("Java", 90, 240); // 绘制文本
68 }
83 // 鼠标事件响应函数
69 public void mouseClicked(MouseEvent e) {
70     ruleIndex++; // 循环改变ruleIndex值
71     ruleIndex %= 12;
72     repaint(); // 重绘
73 }
74
75 public void mousePressed(MouseEvent e) {
76 }
77 public void mouseReleased(MouseEvent e) {
78 }
79 public void mouseEntered(MouseEvent e) {
80 }
81 public void mouseExited(MouseEvent e) {
82 }
83 }

```

这个程序显示了12种Porter-Duff合成规则。类Composting是JPanel的派生类，它实现了MouseListener接口。

把AlphaComposite类定义的常量，作为合成规则存放在一个整型数组中（第33行）。变量ruleIndex指向当前的合成规则。mouseClicked方法将ruleIndex的值加1，并把该值对12求余（第71~72行），然后，调用repaint方法刷新画面而显示新的合成规则。因此，每次在面板中点击鼠标的时候，面板会转向一个不同的合成规则。

在CompositePanel的构造函数中，实现从磁盘中读入一个图像文件（第41行）的功能。在paintComponent方法中（第54行），首先绘制图像，然后调用AlphaComposite类中的getInstance静态方法，设置当前的合成规则，并将 α 值设置为0.4，最后绘制一个红色的圆以及一个白色的字符串“Java”。

即时的绘制表面本身并不支持 α 通道，任何像素的 α 值总是隐式地设为1.0。因此，当对象显示在屏幕上时，目标像素的 α 值就变成1.0。由于这一原因，一些合成规则并不能产生有趣的结果。例如，由于目标像素的 α 值为1，规则DstOver、DstOut与Xor总是忽略掉源像素。要得到更有趣的合成效果，可以通过在具有 α 通道的离屏图像上进行绘制来得到。

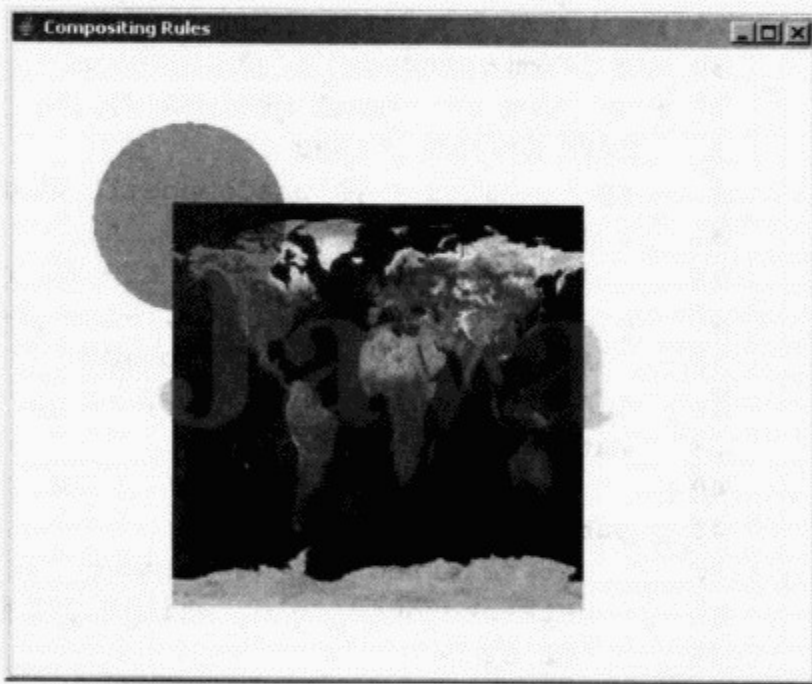


图3-13 使用SRC_OVER规则绘制的重叠物体，该图例展示了12种合成规则之一的合成效果

3.7 裁剪

裁剪路径定义了一个区域，在这个区域中的对象是可见的。Graphics2D维护了一个当前的裁剪区域。当显示一个对象时，根据裁剪路径裁剪该对象，落在裁剪区域之外的部分将不显示

出来。任何Shape对象都可以用于裁剪。下面的代码段将椭圆用做裁剪形状，并在上面绘制一幅图像。只有位于椭圆之内的图像才加以显示。

```
Graphics2D g2 = (Graphics2D)g;  
Shape ellipse = new Ellipse2D.Double(0,0,300,200);  
g2.setClip(ellipse);  
g2.drawImage(image,0,0,this);
```

Graphics2D类中另外一个改变裁剪区域的方法是：

```
void clip(Shape path)
```

这个方法用给定的形状进一步裁剪当前的裁剪区域。

程序清单3-7举例说明了裁剪路径的用法。在下一节将给出另外一个实例。在这个简单例子中，对一个Graphics2D对象使用裁剪路径创建一个特殊的形状，用它来对随后的绘制进行裁剪。图3-14给出了这个程序的一次运行结果。

程序清单3-7 TestClip.java

```
1 package chapter3;  
2  
3 import java.awt.*;  
4 import javax.swing.*;  
5 import java.awt.geom.*;  
6 //定义TestClip类，继承自JApplet类，演示使用裁剪路径  
7 public class TestClip extends JApplet {  
8     public static void main(String s[]) {  
9         JFrame frame = new JFrame();//生成主窗口  
10        frame.setTitle("Clip Path");//设置窗口标题栏  
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理  
12        JApplet applet = new TestClip();//创建TestClip对象  
13        applet.init();//初始化  
14        frame.getContentPane().add(applet);//添加applet到主窗口内容窗格  
15        frame.pack();  
16        frame.setVisible(true);//设置窗口可见  
17    }  
18    //重写JApplet初始化方法  
19    public void init() {  
20        JPanel panel = new ClipPanel();//创建ClipPanel对象  
21        getContentPane().add(panel);//加入TestClip对象内容窗格  
22    }  
23 }  
24 //定义ClipPanel类，继承自JPanel类  
25 class ClipPanel extends JPanel {  
26     public ClipPanel() {  
27         setPreferredSize(new Dimension(500, 500));//设置组件首选大小  
28         setBackground(Color.white);//设置背景颜色为白色  
29     }  
30     //重写组件绘制方法  
31     public void paintComponent(Graphics g) {  
32         super.paintComponent(g);  
33         Graphics2D g2 = (Graphics2D)g;  
34         GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);  
35         path.moveTo(100,200);//生成裁剪路径  
36         path.quadTo(250, 50, 400, 200);  
37         path.lineTo(400,400);  
38         path.quadTo(250,250,100,400);
```



```

39     path.closePath();
40     g2.clip(path); //设置裁剪区域
41     g2.setColor(new Color(200,200,200)); //设置颜色
42     g2.fill(path); //填充
43     g2.setColor(Color.black);
44     g2.setFont(new Font("Serif", Font.BOLD, 60));
45     g2.drawString("Clip Path Demo", 80, 200); //绘制字符串
46     g2.drawOval(50, 250, 400, 100); //绘制椭圆
47 }
48 }

```

这个程序和前面的例子有相似的结构。作为一个同时具有main函数的applet，该程序既可以作为一个应用程序，也可以作为applet运行。ClipPanel类是JPanel类的派生类，用于形成程序所需的画布。

在paintComponent方法(第31行)中，定义了一个具有两条直线段和两条二次曲线的GeneralPath对象。通过调用clip方法，将这个封闭路径设置为Graphics2D对象的当前裁剪路径。为了以浅灰色阴影方式显示裁剪区域，在fill方法中再次用到了路径。本例中绘制了两个图形对象：一个文本字符串“Clip Path Demo”和一个椭圆。裁剪操作所带来的效果是很明显的，它只使位于裁剪路径之内的对象部分才是可见的。

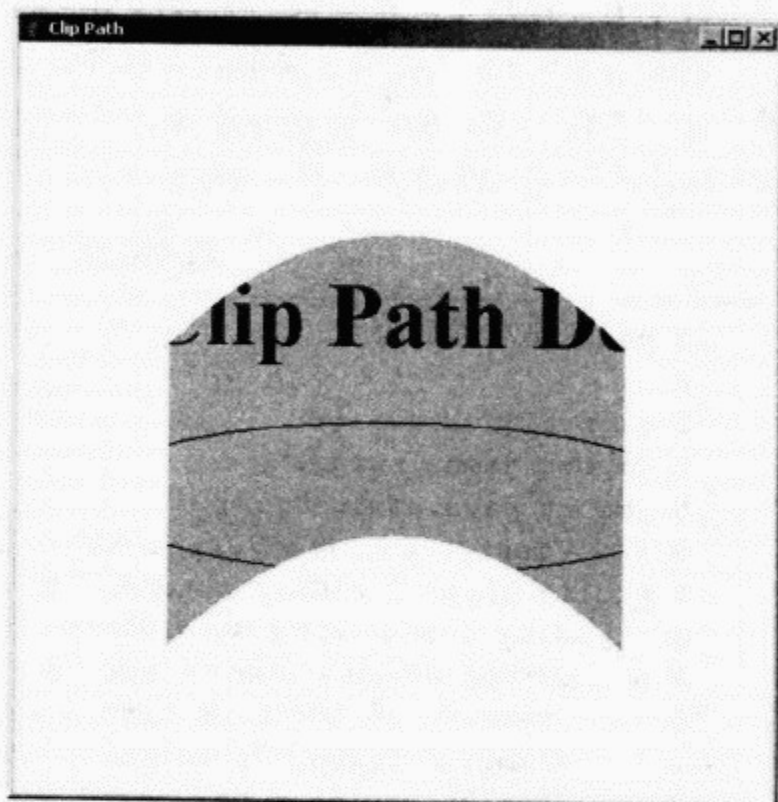


图3-14 裁剪路径将灰色区域包围起来，并完成对图形的裁剪

3.8 文本和字体

在计算机图形学中，文本表示一种特殊类型的几何对象。文本字符串可以通过具有标准编码格式的字符序列紧凑地表示，这样的编码格式有ASCII码和Unicode等。字符的实际绘制形状由预定义的字体决定。字符形状的几何描述称作字形 (glyph)。一种字体是整个字母表中所有字形的集合。

注意 字符和字形的关系并不总是一对一的。有时一种字形可以对应几个字符，比如连体 (ligature) 的情况就是如此。当字体中某两个字符连在一起绘制时，连体就发生了。在一些字体中，常见的连体字是“fi”，如图3-15所示。

Java 2D提供了丰富的字体和文本操作功能。最常用的高层文本使用方法，包括创建一个Font对象，以及调用Graphics2D中的setFont和drawString方法。

可以使用如下的构造函数创建Font对象：

```

//Font的构造函数
Font(String name, int style, int size)

```

name参数描述了字体的字体名或逻辑名字。一个字体由该字体的字体名标识，如“Times New Roman”。在一个环境中，可用的字体是平台相关的。Java也支持逻辑字体 (logical font) 以增强可

fi

图3-15 常见的一种连体

移植性，一个逻辑字体映射为特定系统中的物理字体。例如，逻辑字体“ScanSerif”映射为Windows系统中的“Arial”字体。Java支持如下五种逻辑字体系统：

```
Serif
SansSerif
Monospaced
Dialog
DialogInput
```

87

style参数是一个选择字体风格（font style）的掩码。在Font中定义了三个位掩码，它们能够通过使用位操作符OR“|”连在一起：

```
PLAIN
ITALIC
BOLD
```

size参数描述了字体的点的大小。

在Graphics2D对象中，可以用下面的方法选择Font对象：

```
void setFont(Font font)
```

在随后对绘制文本的方法调用中，选用的新字体将生效：

```
void drawString(String s, int x, int y)
void drawString(String s, float x, float y)
```

除了系统中现有字体之外，可以通过修改现有字体的一些属性而衍生出新的字体。比如说，下列Font类中的方法产生衍生字体（derived font）：

```
//衍生新字体的方法
Font deriveFont(int style)
Font deriveFont(float size)
Font deriveFont(int style, float size)
Font deriveFont(AffineTransform tx)
Font deriveFont(int style, AffineTransform tx)
Font deriveFont(Map attributes)
```

字体点阵的大小只是对绘制的文本大小提供一种粗略的表示。所绘制文本的实际大小通常与字符串包含的字符有关。在文本定位等应用当中，知道文本的实际几何尺寸是很有用处的。字体标度（font metric）用来度量根据特定字体绘制的文本的大小。Font类通过下面的方法提供字体度量信息：

```
//获取字体标度信息
Rectangle2D getStringBounds(String str, FontRenderContext frc)
LineMetrics getLineMetrics(String str, FontRenderContext frc)
```

由于度量精度依赖于绘制选项，所以上面的方法使用FontRenderContext对象获取额外的信息。FontRenderContext对象可以通过Graphics2D类的方法获得：

```
//获取字体绘制上下文
FontRenderContext getFontRenderContext()
```

getStringBounds方法返回字符串的边界矩形。getLineMetrics方法返回一个LineMetrics对象，该对象包含更多的细节性直线标度数据。基线（baseline）是一种字体的参照线，上升量（ascent）是字体在基线之上的量，下降量（descent）是字体在基线之下的量，行距（leading）是在两行文本之间外加的空间。下列LineMetrics类的方法就可以用来获取标度信息：

```
float getAscent()
float getDescent()
float getLeading()
```


程序清单3-8展示了与字体相关的特征。这个例子实际展示了衍生字体和字体标度的使用情况。图形中一共显示了三行文本，第一行文本是用向左倾斜的衍生字体进行绘制的，第二行将文本串连同它的边界矩形一起进行显示，第三行显示了文本的基线、上升量、下降量与行距。图3-16给出了程序的一次运行结果实例。

88

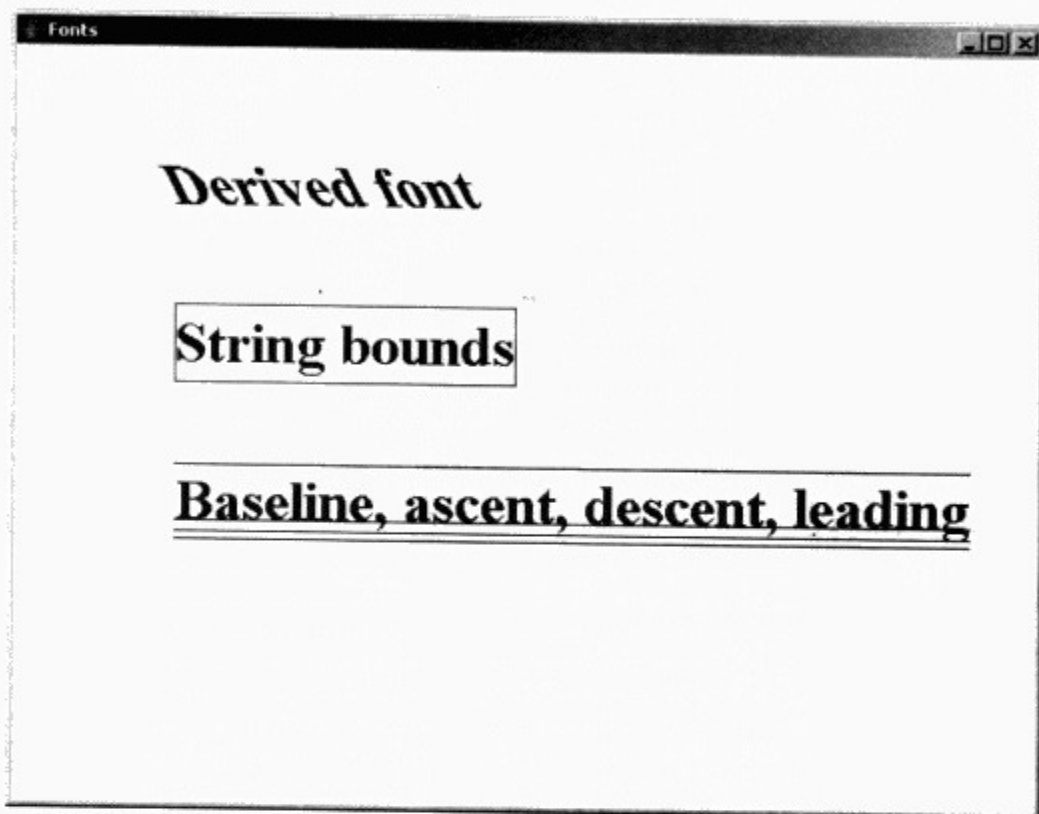


图3-16 第一行显示了使用衍生字体绘制的文本，第二行绘制了边界矩形，第三行给出了基线以及相应的上升量、下降量和行距

程序清单3-8 FontFun.java

```

1 package chapter3;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.geom.*;
6 import java.awt.font.*;
7 //定义FontFun类，继承自JApplet类，演示与字体相关的特性
8 public class FontFun extends JApplet {
9     public static void main(String s[]) {
10         JFrame frame = new JFrame();//生成主窗口
11         frame.setTitle("Fonts");//设置标题栏
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
13         JApplet applet = new FontFun(); //创建FontFun对象
14         applet.init();//初始化
15         frame.getContentPane().add(applet); //添加applet到frame内容窗格
16         frame.pack();
17         frame.setVisible(true);//设置窗口可见
18     }
19     //重写JApplet初始化方法
20     public void init() {
21         JPanel panel = new FontPanel();//创建FontPanel对象
22         getContentPane().add(panel);//添加到FontFun对象内容窗格
23     }
24 }
25 //定义FontPanel类，继承自JPanel类

```

```
26 class FontPanel extends JPanel {
27     public FontPanel() {
28         setPreferredSize(new Dimension(640, 480)); // 设定组件首选大小
29         setBackground(Color.white); // 设定背景颜色为白色
30     }
31     // 重写组件绘制方法
32     public void paintComponent(Graphics g) {
33         super.paintComponent(g);
34         Graphics2D g2 = (Graphics2D)g;
35         Font font = new Font("Serif", Font.BOLD, 36); // 创建一种字体
36         AffineTransform tx = new AffineTransform(); // 设置变换
37         tx.shear(0.5, 0);
38         g2.setFont(font.deriveFont(tx)); // 设置为根据变换导出的字体
39         g2.drawString("Derived font", 100, 100); // 根据指定字体绘制字符串
40
41         g2.setFont(font); // 重新设置字体
42         FontRenderContext frc = g2.getFontRenderContext(); // 获得文本绘制上下文对象
43         String str = "String bounds";
44         Rectangle2D bounds = font.getStringBounds(str, frc); // 获取字符串边界矩形
45         g2.translate(100, 200); // 平移
46         g2.draw(bounds); // 绘制矩形边界
47         g2.drawString(str, 0, 0); // 绘制字符串
48         // 绘制字符串的基线、上升量、下降量和行距
49         str = "Baseline, ascent, descent, leading";
50         g2.translate(0, 100); // 平移
51         int w = (int)font.getStringBounds(str, frc).getWidth();
52         LineMetrics lm = font.getLineMetrics(str, frc);
53         g2.drawLine(0, 0, w, 0);
54         int y = -(int)lm.getAscent();
55         g2.drawLine(0, y, w, y);
56         y = (int)lm.getDescent();
57         g2.drawLine(0, y, w, y);
58         y = (int)(lm.getDescent() + lm.getLeading());
59         g2.drawLine(0, y, w, y);
60
61         g2.drawString(str, 0, 0); // 绘制字符串
62     }
63 }
```

该程序用 FontPanel 类的 paintComponent 方法绘制了三行文本（第32行）。创建的一个 AffineTransform 对象，用于执行水平方向上的错切变换。对一个点阵大小为36的Serif粗体字体进行这个错切变换，就衍生出一种新字体。由于使用的是错切变换，因此衍生的字体向左倾斜。在绘制文本字符串“Derived font”时，用到了这个衍生字体。

通过 Graphics2D 对象可以获得 FontRenderContext 对象（第42行）。使用 Font 对象的 getStringBounds 方法获取字符串“String bounds”的边界矩形，并将字符串及其边界矩形一起绘制出来。

第三行文本是字符串“Baseline、ascent、descent、leading”。文本的基线是根据文本的宽度绘制出来的。使用 getLineMetrics 方法，获取该文本字符串的 LineMetrics 对象（第52行）。从该对象中获取上升量、下降量和行距的值，然后根据这些值绘制出相对于基线的其他直线。

Java 2D 还提供了用于字体相关操作的高级函数。特别地，字体的字符字形能够作为一系列 Shape 对象提取出来。这样一来，对字形进行复杂的处理和应用，就可以得到各种各样的视觉效果。Font 类代表一种字体，GlyphVector 类封装了一系列字形的几何描述。使用下面的 Font 方

90 法，可以为某种字体的字符串获取一个GlyphVector对象：

```
//获取GlyphVector对象
GlyphVector createGlyphVector(FontRenderContext frc, String str)
```

FontRenderContext对象定义了绘制一种字体所需要的一些度量值。该对象可以通过调用Graphics2D对象的getFontRenderContext方法得到。一旦创建了GlyphVector对象，就可以使用下面的方法获得字形的Shape对象：

```
//获取字形的Shape对象
Shape getOutline()
Shape getOutline(float x, float y)
```

x和y参数用于为字形的绘制指定起始位置，返回的相应字形的Shape对象，能够像其他Shape对象一样进行处理和绘制。程序清单3-9演示了用字形作为裁剪形状的使用情况。该程序用到了一种将文本字符串的字形作为Shape对象进行提取的方法，并将该Shape对象作为裁剪路径加以使用。程序的一次试运行的结果如图3-17所示。

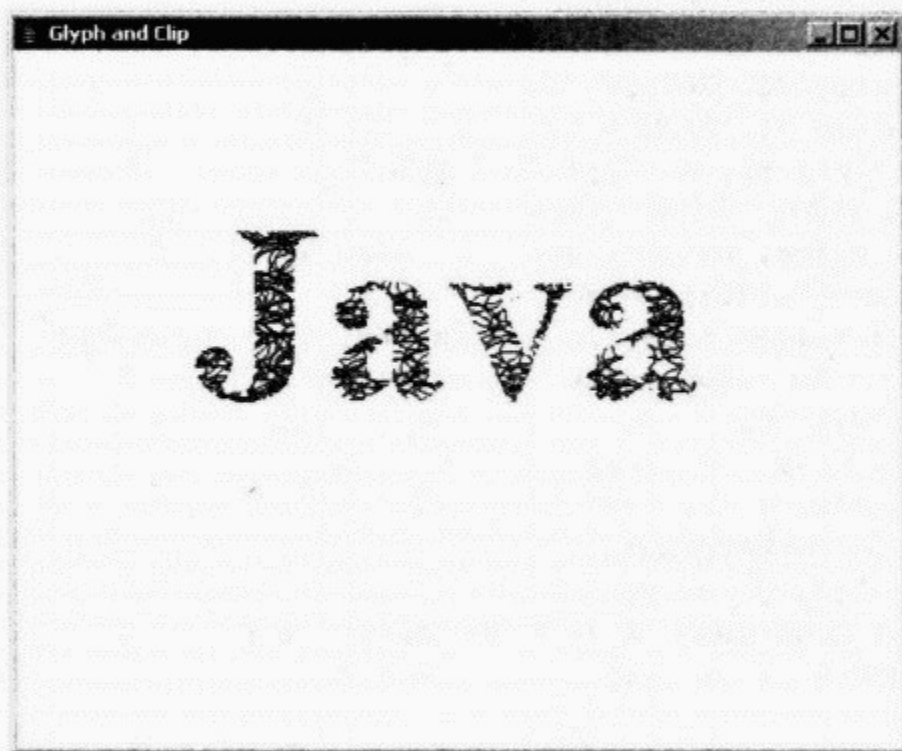


图3-17 在屏幕上随机绘制的2000个椭圆，经过字符串“Java”的字形所定义的裁剪形状进行了裁剪

程序清单3-9 GlyphClip.java

```
1 package chapter3;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.font.*;
7 import java.awt.geom.*;
8 //定义GlyphClip类，继承自JApplet类，演示使用字体轮廓作为裁剪形状
9 public class GlyphClip extends JApplet {
10     public static void main(String s[]) {
11         JFrame frame = new JFrame();//生成主窗口
12         frame.setTitle("Glyph and Clip");//设置窗口标题栏
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
14         JApplet applet = new GlyphClip();//创建GlyphClip对象
15         applet.init();//初始化
```



```
16     frame.getContentPane().add(applet);//添加applet到frame内容窗格
17     frame.pack();
18     frame.setVisible(true);//设置窗口可见
19 }
20 //重写JApplet初始化方法
21 public void init() {
22     JPanel panel = new GlyphPanel();//创建GlyphPanel对象
23     getContentPane().add(panel);//加入GlyphClip内容窗格
24 }
25 }
26 //定义GlyphPanel类,继承自JPanel类
27 class GlyphPanel extends JPanel {
28     public GlyphPanel() {
29         setPreferredSize(new Dimension(500, 400));//设置组件首选大小
30         setBackground(Color.white);//设置背景为白色
31     }
32     public void paintComponent(Graphics g) { //重写组件绘制方法
33         super.paintComponent(g);
34         Graphics2D g2 = (Graphics2D)g;
35         Font font = new Font("Serif", Font.BOLD, 144); //创建字体
36         FontRenderContext frc = g2.getFontRenderContext();//获取字体绘制上下文对象
37         GlyphVector gv = font.createGlyphVector(frc, "Java");//获取字体轮廓向量对象
38         Shape glyph = gv.getOutline(100,200); //根据获取的字形设置裁剪区域
39         g2.setClip(glyph);//设置裁剪区域
40         g2.setColor(Color.red);//设置绘制颜色为红色
41         for (int i = 0; i < 2000; i++) { //随机生成2000个椭圆形状
42             Shape shape = new Ellipse2D.Double(Math.random()*500,
43                 Math.random()*400, 30, 20);
44             g2.draw(shape);//在裁剪区域中绘制椭圆
45         }
46     }
47 }
```

91

这个程序在JPanel的子类中显示了特殊的图形，图形的绘制是通过重载paintComponent方法实现的。该程序还创建了一个点阵大小为144的粗体“Serif”字体对象，同时通过使用Graphics2D对象获得了FontRenderContext对象。

通过字体和字体绘制上下文，获得字符串“Java”的字形信息，并把它们保存在GlyphVector变量中（第37行）。使用getOutline方法，把字形向量转换为Shape对象（第38行）。然后，调用Graphics2D对象的setClip方法，设置用于绘制功能的裁剪路径。

程序在面板上绘制了2000个位于随机位置的椭圆（第41~44行），不过，只有在字形内部的椭圆才是可见的。虽然没有显式地绘制“Java”文本字符串，但是在裁剪路径内部的绘制结果，却非常清晰地突出了该文本字符串的字形轮廓。

主要的类和方法

- java.awt.Color 封装颜色的类。
- java.awt.Paint Color、GradientPaint与TexturePaint类的接口。
- java.awt.GradientPaint 渐变涂色类。
- java.awt.TexturePaint 纹理涂色类。
- java.awt.Stroke 笔划定义接口。
- java.awt.BasicStroke 常用笔划的实现。

- `java.awt.geom.AffineTransform` 封装2D仿射变换的类。
- `javax.awt.AlphaComposite` alpha合成规则类。
- 92 • `java.awt.Graphics2D.setClip(Shape)` 设置当前裁剪路径的方法。
- `java.awt.Graphics2D.setComposite(AlphaComposite)` 设置当前合成规则的方法。
- `java.awt.Font` 封装字体的类。
- `java.awt.font.LineMetrics` 字体标度类。
- `java.awt.font.GlyphVector` 封装了一系列文本字符串字形的类。

关键术语

- 颜色空间 (color space) 使用数值描述颜色的系统。
- 仿射变换 (affine transformation) 保持平行性质的变换。
- 变换组合 (transform composition) 将两个或多个变换组合起来, 形成一个新的变换。
- Porter-Duff 规则 (Porter-Duff rule) 针对已有形状区域而使用, 并采用合并与交集之类的集合运算来创建新几何形状的方法。
- 裁剪路径 (clip path) 用于定义限制绘制区域的形状。
- 字体 (font) 关于字符集的形状设计集合。
- 字体标度 (font metric) 关于所绘制文本的度量值, 比如字体上升量、下降量和行距等。
- 字形 (glyph) 关于特定字体的文本字符串的几何描述。
- 连体字 (ligature) 多个字符经过特殊组合形成一个字形。

本章提要

- 本章讨论了2D图形绘制的一些重要方面, 以及它们在Java 2D中的实现, 涵盖的主要内容包括颜色、笔划、变换、裁剪路径、合成规则和字体等。
- 颜色, 或者更一般地说, 涂色模式, 指的是在绘制可视对象时要用到的属性。Java 2D通过类Color、GradientPaint和TexturePaint提供了三种涂色模式。
- 笔划定义了线型的细节。Java 2D提供了一个Stroke类型的接口, 该接口既可用于一般的笔划定义, 又可用于具体实现一个BasicStroke类来提供线宽、线帽样式及笔划连接类型等常用的笔划属性。
- 仿射变换是一大类变换, 普遍用于对象变换和观察变换。仿射变换保留平行性质。基本的仿射变换包括平移、旋转、缩放、错切和反射。Java 2D对仿射变换提供了全面的支持。AffineTransform类定义了一个仿射变换, 并包含很多用于指定变换的构造函数和方法。createTransformedShape方法为执行对象变换提供了一条途径。仿射变换对象还可以由Graphics 2D对象用来设置观察变换。对变换进行组合, 能够形成更复杂的变换。
- Porter-Duff 规则利用 α 通道定义了合成规则。12条合成规则详细说明了源像素颜色和目标像素颜色的各种合成方法。使用合适的合成规则, 能够获得诸如透明色等有趣的视觉效果。
- 93 • 裁剪是一种可以生成复杂视觉效果的绘制机制。在Java 2D中, 一个Graphics2D对象的裁剪路径能够设置到任何形状的对象上。
- 文本字符串是有用的可视对象。一个字体定义了所有字符集的字形。除了利用drawString方法进行通常的文本绘制外, 文本串的字形值还可以作为Shape对象加以提取。这种Shape对象可以像其他Shape对象一样直接进行操作。

复习题

3.1 可见颜色的波长是多少?

- 3.2 在一个RGB系统中, 如果红、绿、蓝等每个颜色分量都用一个字节表示, 则总共可以形成多少种颜色?
- 3.3 使用float参数构造一个颜色对象, 与下面的对象等价: `new Color(255, 0, 128)`。
- 3.4 使用int参数构造一个颜色对象, 与下面的对象等价: `new Color(0f, 0.5f, 0.125f)`。
- 3.5 构造一个循环类型的渐变涂色模式, 实现从(0,0)点的红色变到(100,100)点的蓝色。
- 3.6 构造一个非循环类型的渐变涂色模式, 实现从(100,0)点的黄色变到(800,600)点的绿色。
- 3.7 绕原点旋转45度角的旋转变换矩阵是什么?
- 3.8 构造一个AffineTransform对象, 实现关于一条过原点直线的一般反射变换。
- 3.9 能够使用仿射变换把椭圆变成圆吗? 所采用的变换是刚体变换吗?
- 3.10 能够使用仿射变换把一个梯形变成正方形吗? 所采用的变换是刚体变换吗?
- 3.11 对于绕原点旋转30度的旋转变换来说, 它的逆变换是什么?
- 3.12 给出一种仿射变换, 用于将x轴变换为y轴, 以及将y轴变换为x轴。
- 3.13 给出绕点 (a, b) 旋转 θ 角的变换矩阵。
- 3.14 求出关于直线 $y = 2x$ 的反射变换矩阵。
- 3.15 求关于直线 $y = 2x - 1$ 的反射变换矩阵。
- 3.16 求沿y轴进行因子为0.5的错切变换的变换矩阵。
- 3.17 求绕原点旋转 $\pi/3$, 然后再关于直线 $y = 2x$ 进行反射变换的复合变换矩阵。
- 3.18 如果源像素的RGB和 α 值分别为0.5、0.0、0.8和0.4, 目标像素的值分别为0.2、1.0、0.5和0.6, 求使用SrcOver合成规则形成的RGB和 α 值。
- 3.19 重新将DstOver规则应用于上述问题。
- 3.20 重新将Src规则应用于上述问题。
- 3.21 重新将Dst规则应用于上述问题。
- 3.22 重新将SrcOut规则应用于上述问题。
- 3.23 重新将DstOut规则应用上述问题。
- 3.24 如果目标颜色的 α 值为1.0, 则哪个合成规则将不受源颜色的影响?
- 3.25 写一段代码, 使用paintComponent方法绘制一个文本字符串的轮廓。

编程练习

- 3.1 编写一个Java程序, 绘制一系列矩形, 这些矩形使用Color类的常量颜色进行填充。
- 3.2 编写一个Java程序, 使用绿颜色绘制如图3-18所示的形状。
- 3.3 编写一个Java程序, 使用在垂直方向从黑色变为白色的渐变涂色方式, 来显示图3-18的形状。
- 3.4 编写一个Java程序, 使用纹理涂色方式显示图3-18的形状。
- 3.5 Swing包含JColorChooser类, 该类允许使用一个对话框交互地选择颜色。修改习题3.2的程序, 使其允许使用JColorChooser类来选择绘制颜色。
- 3.6 绘制一个笔划宽度为20的圆角五边形。
- 3.7 对一个Rectangle2D对象, 使用AffineTransform变换操作, 创建一个位于原点且旋转45度的正方形并显示之。
- 3.8 编写一个Java程序, 显示字符串“Hello 2D”的镜像图像(提示: 使用反射变换)。
- 3.9 编写一个Java程序, 实现关于直线 $y = 2x$ 的反射变换。绘制一个起点在(0, 100), 大小为 100×50 的矩形。对该矩形进行反射变换, 并以不同的颜色绘制变换后的形状。
- 3.10 编写一个Java程序, 用于沿(300, 300)点绘制一个圆形文本, 如图3-19

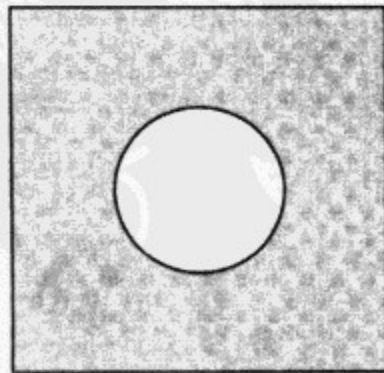


图3-18 一个填充的形状

所示（提示：对每个字符使用drawString方法绘制，并重复运用旋转变换）。

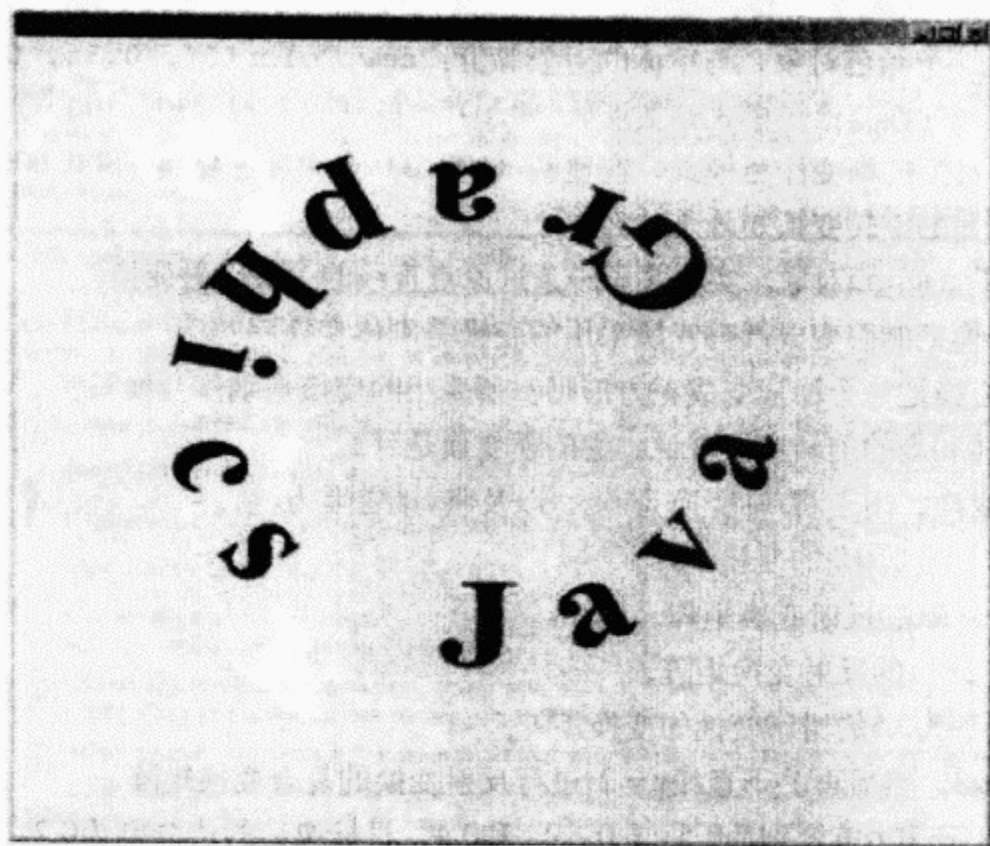


图3-19 一个环形的文本

- 3.11 编写一个Java程序，实现沿直线 $y = x$ 进行一个缩放因子为3的缩放变换。绘制一个起点在 $(0, 0)$ 点、边长为100的正方形。对该正方形进行缩放变换，并以不同的颜色绘制变换后的形状（提示：把变换分解为标准的缩放变换和两个旋转变换）。
- 95 3.12 使用图3-18的形状作为裁剪路径，并使用大字体绘制文本串“Java 2D”。
- 3.13 编写一个程序，用于加载一幅图像，并仅显示图像的一个椭圆区域。使用裁剪路径完成这一效果。
- 3.14 通过旋转45度生成一种新字体，并使用该字体绘制一个字符串。
- 96 3.15 使用字形形状和区域几何模型，显示字符“N”和“Y”经过重叠而形成的字形轮廓。



第4章 2D图形：高级话题（可选）

学习目标

- 理解B样条曲线。
- 构建自定义形状基元。
- 运用基本图像处理技术。
- 生成分形图像。
- 创建2D动画。
- 实现图形打印。

97

4.1 引言

前面的章节讨论了2D计算机图形系统的基本概念和技术，以及Java 2D包。本章将讨论2D图形学方面一些更高级的话题，以及Java 2D没有提供的一些特性。

样条曲线是计算机图形学中重要的建模工具。B样条曲线是一种平滑的曲线，它通过一系列控制点来定义。Java 2D对于绘制样条曲线并没有提供直接的支持。但是，一段B样条曲线可以转化为多段贝塞尔曲线。在本章将介绍这方面的技术。

Java 2D通过一组实现了Shape接口的类层次结构，提供了一些常用的基元。同样，用户可以实现自己的基元。本章将介绍一种实现自定义Shape类的技术，该类能够像内建类一样传递给Graphics2D对象进行绘制。

尽管图像处理是一个独立的范畴，但它与计算机图形学也有着紧密的联系。就像之前在纹理填充例子中所看到的那样，图像是计算机图形学中有用的对象。本章将介绍Java 2D中的图像操作特性。Java 2D提供了一个图像模型，它大大改进了以前的AWT模型。在Java API的支持下，可以方便地对图像进行加载、处理和保存。从零开始创建一幅图像也是可能的。本章使用分形图像来说明Java中的图像创建过程。

动画可以用来创建一个动态场景中的一系列图像。它在图形系统中新加入了一个时间维度。Java语言的多线程能力，对于在Java 2D程序中实现动画提供了至关重要的支持。在这方面将介绍一些实例，包括介绍细胞自动机在内。

图形打印是许多图形化应用的一部分。Java 2D提供了便捷的打印支持，这和在屏幕上绘制图形非常一致。有关Java如何实现图形打印方面的内容，将在后面进行讨论。

4.2 样条曲线

样条曲线由一序列经过平滑连接的多项式曲线组成。B样条曲线是一类在CAD（计算机辅助设计）和其他的计算机图形应用中得到广泛使用的样条曲线。特别地，三次B样条曲线是计算机图形学中使用得最为普遍的B样条曲线。

贝塞尔曲线的数学定义，可以由参数方程给出。一条控制点为 p_0, p_1, \dots, p_n 的一般 n 次贝塞尔曲线（general Bézier curve），由如下方程进行定义：

$$s(t) = \sum_{i=0}^n p_i B_{ni}(t)$$

此处的 $B_{n,i}(t)$ 为Bernstein多项式或Bernstein基 (Bernstein basis)。

$$B_{n,i}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Java 2D所支持的二次和三次曲线, 分别是Bézier曲线在次数 $n=2$ 和 $n=3$ 时的特例。它们可以表达成如下的方程形式:

$$\begin{aligned} s_2(t) &= (1-t)^2 p_0 + 2t(1-t)p_1 + t^2 p_2 \\ s_3(t) &= (1-t)^3 p_0 + 3t(1-t)^2 p_1 + 3t^2(1-t)p_2 + t^3 p_3 \end{aligned}$$

98

一条B样条曲线 (B-spline curve) 由一组控制点序列来定义。像Bézier曲线一样, B样条曲线沿着控制点序列所形成的大致走向不断延伸, 但它并非总是经过这些控制点。一条一般的 k 次B样条曲线由 $n+1$ 个控制点 p_0, p_1, \dots, p_n , 以及一序列称为节点的 $n+k+2$ 个参数值进行定义, 这些参数值是 $t_0 \leq t_1 \leq \dots \leq t_{n+k+1}$ 。B样条曲线的参数方程可以表示成如下形式:

$$p(t) = \sum_{i=0}^n p_i N_{k,i}(t)$$

该曲线仅仅在区间 $[t_3, t_{n+k-2})$ 上有定义。函数 $N_{k,i}(t)$ 称为归一化的B样条调和函数 (normalized B-spline blending function), 其递归定义形式为:

$$\begin{aligned} N_{0,i}(t) &= \begin{cases} 1, & t \in [t_i, t_{i+1}) \\ 0, & \text{其他值} \end{cases} \\ N_{k,i}(t) &= \frac{t-t_i}{t_{i+k}-t_i} N_{k-1,i}(t) + \frac{t_{i+k+1}-t}{t_{i+k+1}-t_{i+1}} N_{k-1,i+1}(t) \end{aligned}$$

B样条曲线是通用的建模工具。曲线的平滑度和连续性可以通过节点来控制。当相邻节点的差值为常数 $t_{i+1}-t_i=c$ 时, 曲线称为均匀B样条曲线 (uniform B-spline)。一般来讲, B样条曲线都是非均匀的。B样条方程也可以将 w 个分量都一样的调和函数应用于齐次坐标系中。在齐次坐标系中表示控制点时, 所构造的B样条曲线称为有理B样条曲线 (rational B-spline curve)。据此, 将最一般的B样条曲线族称为NURBS (nonuniform rational B-spline, 非均匀有理B样条曲线)。

在这一节中, 仅仅考虑三次B样条曲线的一个特殊类型, 它是三次Bézier曲线的直接扩展形式。在这种样条形式中, 除了将起始4个节点和最后4个节点的值设定为相等之外, 其余的节点值都是均匀分布的:

$$\begin{aligned} t_0 &= t_1 = t_2 = t_3 \\ t_{i+1} - t_i &= 1, i = 3, 4, \dots, n \\ t_{n+1} &= t_{n+2} = t_{n+3} = t_{n+4} \end{aligned}$$

重复的结点值使曲线通过第一个和最后一个控制点, 这与Bézier曲线类似。事实上, 如果 $n=3$, 一条这种类型的三次B样条曲线恰好就是一条规则的三次Bézier曲线:

$$\begin{aligned} N_{0,0}(t) &= N_{0,1}(t) = N_{0,2}(t) = 0, N_{0,3}(t) = \chi_{0,1}, N_{0,4}(t) = N_{0,5}(t) = N_{0,6}(t) = 0 \\ N_{1,0}(t) &= N_{1,1}(t) = 0, N_{1,2}(t) = (1-t)\chi_{0,1}, N_{1,3}(t) = t\chi_{0,1}, N_{1,4}(t) = N_{1,5}(t) = 0 \\ N_{2,0}(t) &= 0, N_{2,1}(t) = (1-t)^2 \chi_{0,1}, N_{2,2}(t) = 2t(1-t)\chi_{0,1}, N_{2,3}(t) = t^2 \chi_{0,1}, N_{2,4}(t) = 0 \\ N_{3,0}(t) &= (1-t)^3 \chi_{0,1}, N_{3,1}(t) = 3t(1-t)^2 \chi_{0,1}, N_{3,2}(t) = 3t^2(1-t)\chi_{0,1}, N_{3,3}(t) = t^3 \chi_{0,1} \end{aligned}$$

用 $\chi_{0,1}$ 表示区间 $[0,1)$ 上的特征函数, 那么该B样条曲线的参数方程就成为:

$$p(t) = (1-t)^3 p_0 + 3t(1-t)^2 p_1 + 3t^2(1-t)p_2 + t^3 p_3, t \in [0, 1)$$

99

这正是三次Bézier曲线的方程。当 $n>3$ 时，B样条曲线有不只一个多项式曲线段，并且其控制点数目多于Bézier曲线的控制点数目。

Java 2D并不直接支持B样条曲线。但是，一条三次B样条曲线可以转化成一系列三次Bézier曲线，从而可以使用Java 2D的三次Bézier曲线支持功能来绘制该B样条曲线。设 p_0, p_1, \dots, p_n 为B样条曲线的控制点，B样条曲线的每一段都可以转化成一条三次Bézier曲线。设Bézier曲线的控制点为 b_0, b_1, b_2, b_3 ，则除第一段和最后一段外，其他的转化关系都可以由下列公式给出：

$$\begin{aligned} b_{-1} &= (p_{i-1} + 2p_i)/3 \\ b_1 &= (2p_i + p_{i+1})/3 \\ b_0 &= (b_{-1} + b_1)/2 \\ b_2 &= (p_i + 2p_{i+1})/3 \\ b_4 &= (2p_{i+1} + p_{i+2})/3 \\ b_3 &= (b_2 + b_4)/2 \end{aligned}$$

第一段和最后一段曲线的转化，要使用不同的处理方式，因为第一个和最后一个控制点是曲线的端点。第一段曲线的转化由下述公式给出：

$$\begin{aligned} b_0 &= p_0 \\ b_1 &= p_1 \\ b_2 &= (p_1 + p_2)/2 \\ b_4 &= (2p_2 + p_3)/3 \\ b_3 &= (b_2 + b_4)/2 \end{aligned}$$

最后一段使用下面的公式：

$$\begin{aligned} b_{-1} &= (2p_{n-2} + p_{n-3})/3 \\ b_1 &= (p_{n-1} + p_{n-2})/2 \\ b_0 &= (b_{-1} + b_1)/2 \\ b_2 &= p_{n-1} \\ b_3 &= p_n \end{aligned}$$

程序清单4-1演示了B样条曲线的转化和绘制情况。在这个例子中，一条B样条曲线转化成了一系列Bézier曲线，它们由GeneralPath进行表示。该例子实现了一个简单的画图程序，让用户可以通过鼠标来输入控制点。B样条曲线和它的控制点都进行了显示，程序的运行样例如图4-1所示。

100

程序清单4-1 BSpline.java

```
1 package chapter4;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.event.*;
6 import java.util.*;
7 import javax.swing.*;
8 //定义BSpline类，继承自JApplet类，演示B样条曲线绘制
9 public class BSpline extends JApplet {
10     public static void main(String s[]) {
11         JFrame frame = new JFrame();//创建主窗口
12         frame.setTitle("B-Spline");//设置窗口标题栏
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
```



```

14     JApplet applet = new BSpline();//生成Bspline实例
15     applet.init();//初始化
16     frame.getContentPane().add(applet);//将applet加入主窗口内容窗格
17     frame.pack();
18     frame.setVisible(true);//设置显示
19 }
20 //重写JApplet初始化方法
21 public void init() {
22     JPanel panel = new BSplinePanel();//创建BsplinePanel实例
23     getContentPane().add(panel); //添加到BSpline内容窗格
24 }
25 }
26 //定义BsplinePanel类, 继承自JPanel类, 实现鼠标事件侦听器
27 class BSplinePanel extends JPanel
28     implements MouseListener, MouseMotionListener {
29     Vector points = null;
30     boolean completed = true;
31
32     public BSplinePanel() {
33         setPreferredSize(new Dimension(640, 480));//设定组件首选大小
34         setBackground(Color.white);//设定背景为白色
35         addMouseListener(this);//添加侦听器
36         addMouseMotionListener(this);
37         points = new Vector();//存储B样条曲线的控制点向量对象
38     }
39     //重写组件绘制方法
40     public void paintComponent(Graphics g) {
41         super.paintComponent(g);
42         Graphics2D g2 = (Graphics2D)g;
43         Point p0 = null;//p0-p4为B样条曲线的控制点
44         Point p1 = null;
45         Point p2 = null;
46         Point p3 = null;
47         float x1,y1,x2,y2,x3,y3, x4, y4;//存放转化后的控制点
48         Iterator it = points.iterator();
49         if (it.hasNext()) {
50             p1 = (Point)(it.next());
51         }
52         while (it.hasNext()) {
53             p2 = (Point)(it.next());
54             g2.drawLine(p1.x, p1.y, p2.x, p2.y);//相邻控制点之间连线
55             p1 = p2;
56         }
57
58         GeneralPath spline = new GeneralPath();
59         int n = points.size();
60         if (n == 0) return;
61         p1 = (Point)points.get(0);
62         spline.moveTo(p1.x, p1.y);
63         g2.drawRect(p1.x-3, p1.y-3, 6, 6);
64         p1 = (Point)points.get(1);
65         p2 = (Point)points.get(2);
66         p3 = (Point)points.get(3);
67         x1 = p1.x;
68         y1 = p1.y;

```

```
69     x2 = (p1.x + p2.x)/2.0f;
70     y2 = (p1.y + p2.y)/2.0f;
71     x4 = (2.0f*p2.x+p3.x)/3.0f;
72     y4 = (2.0f*p2.y+p3.y)/3.0f;
73     x3 = (x2+x4)/2.0f;
74     y3 = (y2+y4)/2.0f;
75     spline.curveTo(x1, y1, x2, y2, x3, y3); //构建该段转化成的B样条曲线
76     g2.drawRect((int)x1-3, (int)y1-3, 6, 6);
77     g2.drawRect((int)x2-3, (int)y2-3, 6, 6);
78     g2.drawRect((int)x3-3, (int)y3-3, 6, 6);
79     for (int i = 2; i < n - 4; i++) {
80         p1 = p2;
81         p2 = p3;
82         p3 = (Point)points.get(i+2);
83         x1 = x4;
84         y1 = y4;
85         x2 = (p1.x+2.0f*p2.x)/3.0f;
86         y2 = (p1.y+2.0f*p2.y)/3.0f;
87         x4 = (2.0f*p2.x+p3.x)/3.0f;
88         y4 = (2.0f*p2.y+p3.y)/3.0f;
89         x3 = (x2+x4)/2.0f;
90         y3 = (y2+y4)/2.0f;
91         spline.curveTo(x1,y1,x2,y2,x3,y3);
92         g2.drawRect((int)x1-3, (int)y1-3, 6, 6);
93         g2.drawRect((int)x2-3, (int)y2-3, 6, 6);
94         g2.drawRect((int)x3-3, (int)y3-3, 6, 6);
95     }
96     p1 = p2;
97     p2 = p3;
98     p3 = (Point)points.get(n-2);
99     x1 = x4;
100    y1 = y4;
101    x2 = (p1.x+2.0f*p2.x)/3.0f;
102    y2 = (p1.y+2.0f*p2.y)/3.0f;
103    x4 = (p2.x+p3.x)/2.0f;
104    y4 = (p2.y+p3.y)/2.0f;
105    x3 = (x2+x4)/2.0f;
106    y3 = (y2+y4)/2.0f;
107    spline.curveTo(x1,y1,x2,y2,x3,y3);
108    g2.drawRect((int)x1-3, (int)y1-3, 6, 6);
109    g2.drawRect((int)x2-3, (int)y2-3, 6, 6);
110    g2.drawRect((int)x3-3, (int)y3-3, 6, 6);
111    p2 = p3;
112    p3 = (Point)points.get(n-1);
113    x1 = x4;
114    y1 = y4;
115    x2 = p2.x;
116    y2 = p2.y;
117    x3 = p3.x;
118    y3 = p3.y;
119    spline.curveTo(x1,y1,x2,y2,x3,y3);
120    g2.drawRect((int)x1-3, (int)y1-3, 6, 6);
121    g2.drawRect((int)x2-3, (int)y2-3, 6, 6);
122    g2.drawRect((int)x3-3, (int)y3-3, 6, 6);
123    g2.draw(spline);
```



```
124     }
125     public void mouseClicked(MouseEvent ev) {
126     }
127
128     public void mouseEntered(MouseEvent ev) {
129     }
130
131     public void mouseExited(MouseEvent ev) {
132     }
133     //鼠标按下事件响应函数
134     public void mousePressed(MouseEvent ev) {
135         Graphics g = getGraphics();
136         if (completed) {
137             points.clear();//清除points向量
138             completed = false;//设置结束标识符，可以重新开始新的曲线
139         }
140         if (ev.getClickCount() == 1) {
141             Point p = ev.getPoint();//取得当前鼠标点击的位置
142             points.add(p);//添加控制点到points向量
143             g.fillOval(p.x-3, p.y-3, 6, 6);//绘制控制点
144         }
145     }
146     //鼠标释放事件响应函数
147     public void mouseReleased(MouseEvent ev) {
148         if (ev.getClickCount() > 1) {
149             completed = true;//设置结束标识符
150             repaint();//重绘
151         }
152     }
153
154     public void mouseMoved(MouseEvent ev) {
155     }
156
157     public void mouseDragged(MouseEvent ev) {
158     }
159 }
```

注意 上面给出的B样条曲线转化情形，仅仅是针对本节所讨论的简单B样条曲线类型的。更一般类型的节点非均匀B样条曲线，同样可以转化为Bézier曲线。

BSplinePanel类继承了JPanel类，并提供了一个面板来绘制B样条曲线。向量points（第29行）用来储存B样条曲线的控制点，鼠标事件用于处理控制点的输入。点击鼠标一次就定义一个控制点，该控制点显示成一个实心小圆点。双击鼠标一次定义曲线的最后一个控制点，同时结束所有控制点的输入。为了简便起见，在任何时刻只能定义一条B样条曲线。在所有的控制点输入以后，就用paintComponent方法来绘制B样条曲线。

从B样条曲线到Bézier曲线的转化过程，在paintComponent方法中实现。该转化使用了本章所介绍的公式。一个GeneralPath对象用来记录转化所得的Bézier曲线序列（第58行）。在所有曲线段都转化完毕后，路径就绘制出来了。控制点所形成的控制多边形，通过一系列的线段显示出来，图中的小方块表示转化过后的Bézier曲线的控制点。

[103]

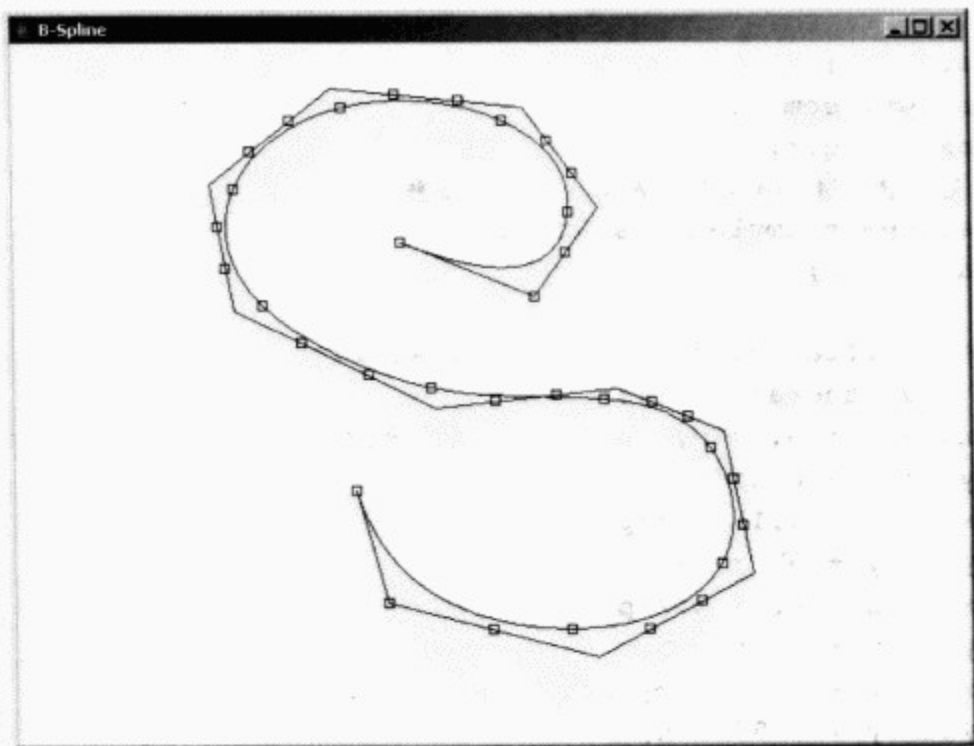


图4-1 通过一系列Bézier曲线显示的一条B样条曲线。多边形表示B样条曲线的控制点，小方块表示了Bézier曲线控制点的位置。

4.3 自定义基元

正如前面几章中所叙述的那样，在Shape家族中定义为类的几何基元（geometric primitive），能够以统一的方式来变换和绘制。同样，用户也可以自己定义基元，让它们和内建的类一样运作。整个事情的关键集中在Shape接口之上，该接口有10个抽象方法。

```
public boolean contains(Rectangle2D rect)//测试所给图元是否完全在图形内部
public boolean contains(Point2D point)
public boolean contains(double x,double y)
public boolean contains(double x,double y,double w,double h)
public Rectangle getBounds()//返回边界矩形
public Rectangle2D getBounds2D()
public PathIterator getPathIterator(AffineTransform at)//返回at变换的PathIterator对象
public PathIterator getPathIterator(AffineTransform at,double flatness)
public boolean intersects(Rectangle2D rect)//测试所给图元是否和图形交叉
public boolean intersects(double x,double y,double w,double h)
```

contains()方法测试给定的点或矩形是否完全处于形状内部，intersects()方法测试相交的情况，getBounds()和getBounds2D()方法返回形状的边界矩形。getPathIterator()方法返回PathIterator对象，该对象使用基本的绘制段来描述路径。

继承GeneralPath类似乎是实现自定义基元的一条捷径，因为GeneralPath类实现了所有的Shape方法，并且它使所有的基本绘制函数都可用于路径的绘制。遗憾的是，这个办法并不可行，因为GeneralPath被声明成了一个final类，是不允许进一步作为基类加以使用的。不过，将GeneralPath类封装在自己定义的类中，就可以继续利用GeneralPath所实现的方法了。在Shape接口中定义的所需方法，简单地调用GeneralPath的对应方法就可以实现。程序清单4-2演示了这种方法，该例子通过封装一个GeneralPath对象来构造自定义的Shape，程序利用两条三次曲线构造了一个心形图案（如图4-2所示）。

程序清单4-2 Heart.java

```
1 package chapter4;
```



```
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import javax.swing.*;
6 //定义Heart类, 继承自Shape类, 表示一个心形图案
7 public class Heart implements Shape {
8     GeneralPath path;
9
10    public Heart(float x, float y, float w, float h) {
11        path = new GeneralPath();
12        float x0 = x + 0.5f*w;//p0, 图形正中上面的点
13        float y0 = y + 0.3f*h;
14        float x1 = x + 0.1f*w;//p1, 左部最高点
15        float y1 = y + 0f * h;
16        float x2 = x + 0f * w;//p2, 最左的点
17        float y2 = y + 0.6f * h;
18        float x3 = x + 0.5f * w;//p3, 图形正中下面的点
19        float y3 = y + 0.9f * h;
20        float x4 = x + 1f * w;//p4, 最右的点
21        float y4 = y + 0.6f * h;
22        float x5 = x + 0.9f * w;//p5, 右部最高的点
23        float y5 = y + 0f * h;
24        path.moveTo(x0, y0);
25        path.curveTo(x1, y1, x2, y2, x3, y3);//左半曲线
26        path.curveTo(x4, y4, x5, y5, x0, y0);//右半曲线
27    }
28
29    public boolean contains(Rectangle2D rect) {
30        return path.contains(rect);
31    }
32
33    public boolean contains(Point2D point) {
34        return path.contains(point);
35    }
36
37    public boolean contains(double x, double y) {
38        return path.contains(x, y);
39    }
40
41    public boolean contains(double x, double y, double w, double h) {
42        return path.contains(x, y, w, h);
43    }
44
45    public Rectangle getBounds() {
46        return path.getBounds();
47    }
48
49    public Rectangle2D getBounds2D() {
50        return path.getBounds2D();
51    }
52
53    public PathIterator getPathIterator(AffineTransform at) {
54        return path.getPathIterator(at);
55    }
56
57    public PathIterator getPathIterator(AffineTransform at,
```

```

58 double flatness) {
59     return path.getPathIterator(at, flatness);
60 }
61
62 public boolean intersects(Rectangle2D rect) {
63     return path.intersects(rect);
64 }
65
66 public boolean intersects(double x, double y, double w, double h) {
67     return path.intersects(x, y, w, h);
68 }
69 }

```

Heart类实现了Shape接口，所以它能够像Shape族的其他几何基元一样进行使用。创建一个GeneralPath对象，并保存在变量path之中（第8行）。在Shape接口中定义的10个所需方法，通过调用path对象中的对应方法来实现。

Heart类定义了一个构造函数来设定图形的边界矩形。在构造函数中（第10行），建立了GeneralPath对象，同时定义了心形形状的路径。两条对称的三次曲线定义了心形的左右边界。先计算出控制点，然后使用GeneralPath类中的curveTo方法将曲线建立起来，这两条封闭起来的路径形成了一个可以填充的区域。

在程序清单4-3中，包含了TestHeart这样一个带有main方法的Java小程序（applet），用于测试Heart图元。程序创建了一个JPanel的匿名子类，并将其加入到该applet中。程序面板通过改写paintComponent方法来画一个心形对象，并使用红色对其进行填充。

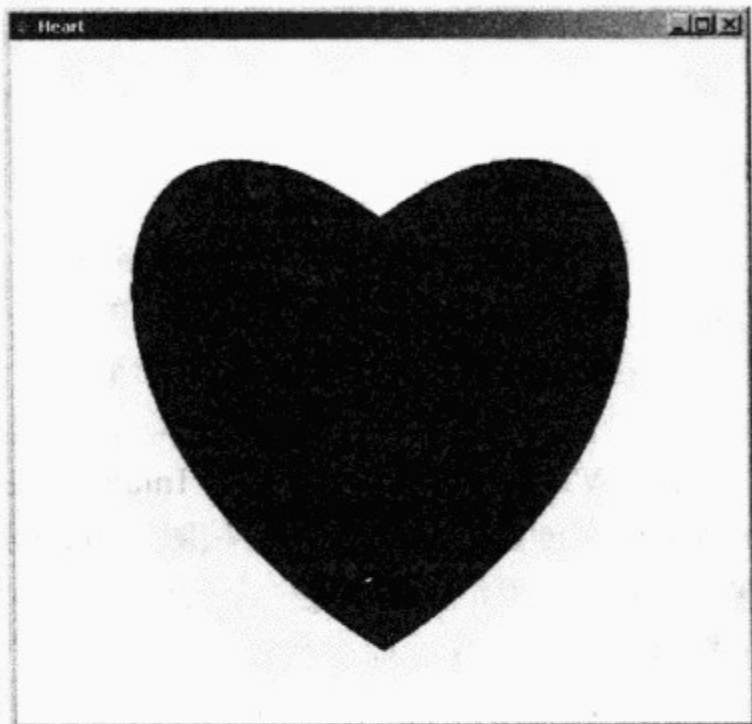


图4-2 采用GeneralPath对象，由两条对称三次曲线构造而成的心形图元

106

程序清单4-3 TestHeart.java

```

1 package chapter4;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.geom.*;
7 //定义TestHeart类，继承自JApplet类，演示使用Heart类
8 public class TestHeart extends JApplet {
9     public static void main(String s[]) {
10         JFrame frame = new JFrame();//创建主窗口
11         frame.setTitle("Heart");//设置窗口标题栏
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
13         JApplet applet = new TestHeart();//创建TestHeart实例
14         applet.init();//初始化
15         frame.getContentPane().add(applet);//添加applet到frame内容窗格
16         frame.pack();
17         frame.setVisible(true);//设置窗口可见
18     }

```



```

19 //重写初始化函数
20 public void init() {
21     JPanel panel = new JPanel() {
22         public void paintComponent(Graphics g) {
23             super.paintComponent(g);
24             Heart h = new Heart(0,0,500,500); //新建Heart对象
25             g.setColor(Color.red);
26             ((Graphics2D)g).fill(h); //用红色填充心形对象
27         }
28     };
29     panel.setBackground(Color.white); //设置背景为白色
30     panel.setPreferredSize(new Dimension(500,500)); //设置组件首选大小
31     getContentPane().add(panel);
32 }
33 }

```

4.4 图像处理

数字化图像是2D图像的一种栅格表示，它通过点值的一个阵列来定义，这些点称为像素(pixel)。每一个像素的值表示对应点的颜色、灰度及其他属性。尽管图像处理是一门具有独立性的专业化学科，但是它同计算机图形学有着紧密的联系。图像是图形绘制中有用的对象，图形绘制的典型输出就是一幅图像。Java 2D提供了强大的图像处理工具。

在AWT中，一幅图像用一个Image类来表示。在AWT中，图像使用“强推”模型(push model)。在创建Image对象时，该图像的数据并不是必需的。生产者是一个实现了ImageProducer接口的对象，而消费者则是一个实现了ImageConsumer接口的对象。生产者作为图像的来源，而消费者从生产者那里取得数据。在一个生产者和一个消费者之间，还可以有一系列同时实现ImageProducer和ImageConsumer接口的过滤器连接成链。生产者以异步形式将数据强推给消费者，消费者不能请求数据。数据传送过程可以通过一个ImageObserver对象实例来监控。这种模型的设计理念，起源于从网络上下载图像的想法。但是，“强推”模型对于图像处理来说并不是很方便。

Java 2D引入了一个新的“即时”模型(immediate model)。新的图像类BufferedImage通过一些立即可用的数据来表示一幅图像。一个BufferedImage类对象包含一个Raster类对象和一个ColorModel类对象。Raster用数字形式表示像素的值，ColorModel则设定了由Raster中的数字值到实际颜色之间的映射关系。

注意 JAI (Java Advanced Image, Java高级图像处理) 是一个可选包，它提供了更高级、更全面的图像处理能力，本书不讨论JAI的相关内容。

一个典型的图像处理循环如图4-3所示。

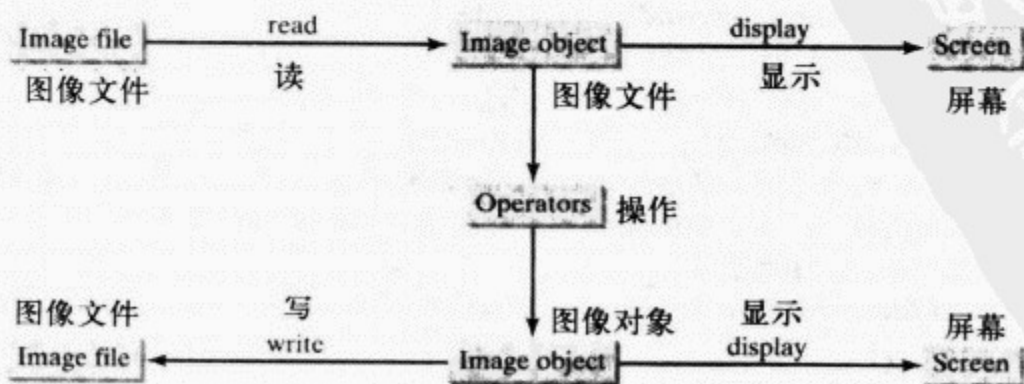


图4-3 一个图像处理循环

源图像通常以众多图像文件格式中的一种来进行表示。Java程序创建图像对象，以表示一幅实际的图像。外部图像文件需要读入图像对象中，图像可能在显示屏或打印机等设备中显示出来。下一步是通过一个或多个图像处理操作来处理图像，所得到的结果是另一个图像对象。处理得到的图像可以显示出来，或者是写入到一个外部的图像文件中。

Java 2D图像处理程序，通常使用BufferedImage类来表示图像。一个BufferedImage对象可以通过它的某一个构造函数来创建。例如，下面的语句创建了一个空白的BufferedImage：

```
BufferedImage bi = new BufferedImage(300,400,BufferedImage.TYPE_INT_RGB);
```

在BufferedImage上绘图时，有必要得到一个Graphics2D对象：

```
Graphics2D g2 = (Graphics2D)(bi.createGraphics());
```

有了Graphics2D对象g2，就可以像在显示屏上绘图一样，在BufferedImage上绘制所有类型的图形。在J2SDK 1.4之前的版本中，标准Java 2D并不支持从外部图像文件或者网络资源中直接载入一个BufferedImage对象。要将一幅图像从文件中载入到Image对象中，可以使用旧的AWT工具，方式如下：

```
Image image = Toolkit.getDefaultToolkit().getImage(imageFileName);
```

在一个applet中，可以使用Applet类中的getImage(url)方法，从网络URL直接载入一幅图像，以此来代替Toolkit对象的使用。

在“强推”模型中，图像的载入是异步的。上述调用会直接返回而不等待图像的载入完成。如果希望确认图像是否完全载入了，可以使用一个MediaTracker对象来强制载入图像，并等待载入的完成。

```
MediaTracker tracker = new MediaTracker(new Component() {});  
tracker.addImage(image,0);  
try{  
    tracker.waitForID(0);  
}catch(InterruptedException ex){  
}
```

MediaTracker的构造函数需要一个Component类的参数。由于Component是一个抽象类，所以不可能创建一个Component类属的实例。作为替代，上述代码使用了一个匿名子类实例newComponent(){}来作为参数。

上面的过程仅仅将一个外部图像文件读入到一个Image对象中。为了将一个由外部载入的Image对象转化成一个BufferedImage对象，可以结合使用Graphics2D对象g2和BufferedImage对象bi。g2对象的drawImage方法允许将对象image画到bi中。

```
g2.drawImage(image,0,0,new Component() {});
```

如果bi的大小与输入图像image的一样，那么上述方法调用的结果，就是把一个Image对象转化成一个BufferedImage对象。drawImage方法的最后一个参数是一个ImageObserver对象。因为Component类实现了ImageObserver接口，所以该匿名子类可以用做一个通用参数。当然，也可以使用null作为ImageObserver参数。如果将这段代码嵌入一个GUI应用中，那么像JPanel对象之类的GUI组件，通常作为图像观测器来使用。使用ImageObserver对象的目的，是为了在原来的AWT“强推”模型中支持图像的异步载入。ImageObserver在Component类中的默认实现形式是重画这个组件，使得在接收到图像数据时，图像能够按增量形式进行显示。

J2SDK 1.4包含了新的ImageIO API，它提供了对于BufferedImage直接读写的支持。要从文件中读入一幅图像，可以简单地调用下面的静态方法：

```
BufferedImage bi = ImageIO.read(file);
```


一旦正确建立了一个BufferedImage对象，就可以对它进行图像处理操作。Java 2D包含了一组用于图像处理的类。这些类实现了BufferedImage接口，如图4-4所示。

RescaleOp通过一个线性函数，对像素值逐个进行重新缩放。一个像素值先乘以一个比例因子，然后再加上一个偏移量。如果 $f(x, y)$ 和 $g(x, y)$ 分别表示处理前后的像素值，则重缩放操作可写成：

$$g(x, y) = af(x, y) + b$$

ColorConvertOp逐个对像素进行颜色转化，该操作可以通过颜色空间来设定。LookupOp基于颜色查找表，逐个对像素值进行转化，该操作可以表示为：

$$g(x, y) = T(f(x, y))$$

AffineTransformOp在图像上进行仿射变换，该操作并不改变一个像素的值，而是将一个像素移动到另一个位置，AffineTransform对象用来设置该变换。仿射变换操作的公式可以表示为：

$$g(x, y) = f(A(x, y))$$

ConvolveOp定义了卷积操作，卷积是一种线性变换。如果一幅图像在数学上可以通过函数 $f(x, y)$ 进行表达，则其对应的卷积操作可以表示为：

$$g(x, y) = \iint K(x - u, y - v) f(u, v) du dv$$

此处， K 是一个固定函数，称之为核（kernel）。卷积的性质由核决定，通过选择适合的核，可以在图像上达到不同的效果，如平滑、锐化或者边缘检测。

对于数字图像，积分变为求和：

$$g(x, y) = \sum_i \sum_j K(x - i, y - j) f(i, j)$$

索引 i, j 遍历整个图像。为提高效率，通常选定的核具有有限的支持域，即 K 只在原点的一个邻域内有非零值。例如， $K(i, j)$ 只有在 $-1 \leq i, j \leq 1$ 的情况下有9个非零值。在这种情况下，上述卷积公式就可以表示为：

$$g(x, y) = \sum_{i=x-1}^{x+1} \sum_{j=y-1}^{y+1} K(x - i, y - j) f(i, j)$$

针对图像上的每一点，卷积的计算只要考虑相邻的9个像素。

在Java 2D中，如果要对BufferedImage对象进行某个操作，只要简单地调用操作对象的filter方法即可。其调用形式如下：

```
dst = op.filter(src, null);
```

BufferedImage可以使用Graphics对象的drawImage方法来显示。例如：

```
public void paintComponent(Graphics g){
    super.paintComponent(g);
    g.drawImage(bi, 0, 0, this);
}
```

这段代码可能出现在一个类似JPanel的组件中。drawImage调用过程将从(0, 0)的位置开始，绘制BufferedImage对象。

在J2SDK 1.4之前，Java 2D并不直接支持将BufferedImage输出到一个外部文件中，或者是

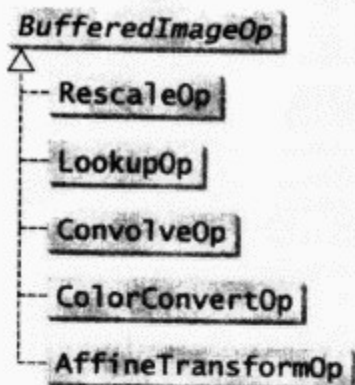


图4-4 BufferedImageOp是一个用于BufferedImage操作的通用接口

将一幅图像编码成一种标准的图像文件格式。新的ImageIO类提供了静态的write方法，用于将图像储存到一个外部文件中，其表示形式如下：

```
ImageIO.write(bi,"png",file);
```

110

第一个参数是需要保存的BufferedImage对象，第二个参数是一个用于限定文件格式的字符串，第三个参数是一个File对象，表示将要写的外部文件。

程序清单4-4展示了包括I/O、处理和显示等操作的完整图像处理程序。用户可以从磁盘文件中载入一幅图像，执行一些普通的图像处理操作，然后将处理过的图像保存到磁盘文件中，运行示例如图4-5所示。

程序清单4-4 ImageProcessing.java

```
1 package chapter4;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.image.*;
6 import java.awt.color.*;
7 import java.awt.geom.*;
8 import java.io.*;
9 import javax.swing.*;
10 import javax.imageio.*;
11 //定义ImageProcessing类，继承自JFrame类，实现ActionListener接口
12 public class ImageProcessing extends JFrame implements
13     ActionListener {
14     public static void main(String[] args) {
15         JFrame frame = new ImageProcessing();//创建主窗口
16         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
17         frame.pack();
18         frame.setVisible(true);//设置窗口可见
19     }
20
21     ImagePanel imageSrc, imageDst;//imageSrc: 原图像, imageDst: 处理过的图像
22     JFileChooser fc = new JFileChooser();//文件打开对话框
23
24     public ImageProcessing() {
25         JMenuBar mb = new JMenuBar();//设置菜单栏
26         setJMenuBar(mb);
27
28         JMenu menu = new JMenu("File");
29         JMenuItem mi = new JMenuItem("Open image");
30         mi.addActionListener(this);
31         menu.add(mi);
32         mi = new JMenuItem("Open image (awt)");
33         mi.addActionListener(this);
34         menu.add(mi);
35         mi = new JMenuItem("Save image");
36         mi.addActionListener(this);
37         menu.add(mi);
38         menu.addSeparator();
39         mi = new JMenuItem("Exit");
40         mi.addActionListener(this);
41         menu.add(mi);
42         mb.add(menu);
```


111

```
43
44     menu = new JMenu("Process");
45     mi = new JMenuItem("Copy");
46     mi.addActionListener(this);
47     menu.add(mi);
48     mi = new JMenuItem("Smooth");
49     mi.addActionListener(this);
50     menu.add(mi);
51     mi = new JMenuItem("Sharpen");
52     mi.addActionListener(this);
53     menu.add(mi);
54     mi = new JMenuItem("Edge");
55     mi.addActionListener(this);
56     menu.add(mi);
57     mi = new JMenuItem("Rescale");
58     mi.addActionListener(this);
59     menu.add(mi);
60     mi = new JMenuItem("Rotate");
61     mi.addActionListener(this);
62     menu.add(mi);
63     mi = new JMenuItem("Gray scale");
64     mi.addActionListener(this);
65     menu.add(mi);
66     mb.add(menu);
67
68     Container cp = this.getContentPane();
69     cp.setLayout(new FlowLayout());
70     imageSrc = new ImagePanel();
71     imageDst = new ImagePanel();
72     cp.add(imageSrc);
73     cp.add(imageDst);
74 }
75 //事件响应函数
76 public void actionPerformed(ActionEvent ev) {
77     String cmd = ev.getActionCommand();
78     if ("Open image".equals(cmd)) { //直接打开图像
79         int retval = fc.showOpenDialog(this);
80         if (retval == JFileChooser.APPROVE_OPTION) {
81             try {
82                 BufferedImage bi = ImageIO.read(fc.getSelectedFile()); //读入图像
83                 imageSrc.setImage(bi);
84                 pack();
85             } catch (IOException ex) {
86                 ex.printStackTrace();
87             }
88         }
89     } else if ("Open image (awt)".equals(cmd)) { //awt工具打开
90         int retval = fc.showOpenDialog(this);
91         if (retval == JFileChooser.APPROVE_OPTION) {
92             Toolkit tk = Toolkit.getDefaultToolkit();
93             Image img = tk.getImage(fc.getSelectedFile().getPath()); //图像读入
94             MediaTracker tracker = new MediaTracker(new Component() {});
95             tracker.addImage(img, 0);
96             try {
97                 tracker.waitForID(0); //等待img读完
98             } catch (InterruptedException ex) {}
99         }
100     }
101 }
```

```

99         BufferedImage bi = new BufferedImage(img.getWidth(this),
100         img.getHeight(this),BufferedImage.TYPE_INT_RGB);
101         bi.getGraphics().drawImage(img, 0, 0, this);
102         imageSrc.setImage(bi);
103     }
104     } else if ("Save image".equals(cmd)) { //保存图像
105         int retval = fc.showSaveDialog(this);
106         if (retval == JFileChooser.APPROVE_OPTION) {
107             try{
108                 ImageIO.write(imageDst.getImage(), "png",
109                 fc.getSelectedFile()); //直接将处理过的图像写入选定文件
110             } catch (IOException ex) {
111                 ex.printStackTrace();
112             }
113         }
114     } else if ("Exit".equals(cmd)) { //退出
115         System.exit(0);
116     } else if ("Copy".equals(cmd)) { //复制
117         imageSrc.setImage(imageDst.getImage()); //将处理过的图像复制到原图像中
118     } else {
119         process(cmd);
120     }
121 }
122
123 void process(String opName) {
124     BufferedImageOp op = null;
125     if (opName.equals("Smooth")) { //平滑
126         float[] data = new float[9];
127         for (int i = 0; i < 9; i++) data[i] = 1.0f/9.0f; //设置3*3的kernel, 元素值均为1/9
128         Kernel ker = new Kernel(3,3,data);
129         op = new ConvolveOp(ker); //卷积
130     } else if (opName.equals("Sharpen")) { //锐化
131         float[] data = {0f, -1f, 0f, -1f, 5f, -1f, 0f, -1f, 0f};
132         Kernel ker = new Kernel(3,3,data);
133         op = new ConvolveOp(ker);
134     } else if (opName.equals("Edge")) { //边缘检测
135         float[] data = {0f, -1f, 0f, -1f, 4f, -1f, 0f, -1f, 0f};
136         Kernel ker = new Kernel(3,3,data);
137         op = new ConvolveOp(ker);
138     } else if (opName.equals("Rescale")) { //重缩放
139         op = new RescaleOp(1.5f, 0.0f, null);
140     } else if (opName.equals("Gray scale")) { //灰度转换
141         op = new ColorConvertOp(ColorSpace.getInstance
142         (ColorSpace.CS_GRAY), null);
143     } else if (opName.equals("Rotate")) { //旋转
144         AffineTransform xform = new AffineTransform(); //新建仿射变换
145         xform.setToRotation(Math.PI/6); //旋转30度
146         op = new AffineTransformOp(xform, AffineTransformOp.
147         TYPE_BILINEAR);
148     }
149     BufferedImage bi = op.filter(imageSrc.getImage(), null); // 处理原图像
150     imageDst.setImage(bi); //传给处理过的图像面板
151     pack();
152 }
153 }
154 //定义ImagePanel类

```



```

155 class ImagePanel extends JPanel {
156     BufferedImage image = null;
157
158     public ImagePanel() {
159         image = null;
160         setPreferredSize(new Dimension(256, 256));
161     }
162
163     public ImagePanel(BufferedImage bi) {
164         image = bi;
165     }
166
167     public void paintComponent(Graphics g) {
168         Graphics2D g2 = (Graphics2D)g;
169         if (image != null)
170             g2.drawImage(image, 0, 0, this); //绘制图像
171         else
172             g2.drawRect(0, 0, getWidth()-1, getHeight()-1); //绘制边框
173     }
174
175     public BufferedImage getImage() {
176         return image;
177     }
178
179     public void setImage(BufferedImage bi) {
180         image = bi;
181         setPreferredSize(new Dimension(bi.getWidth(), bi.getHeight()));
182         invalidate();
183         repaint();
184     }
185 }

```

113

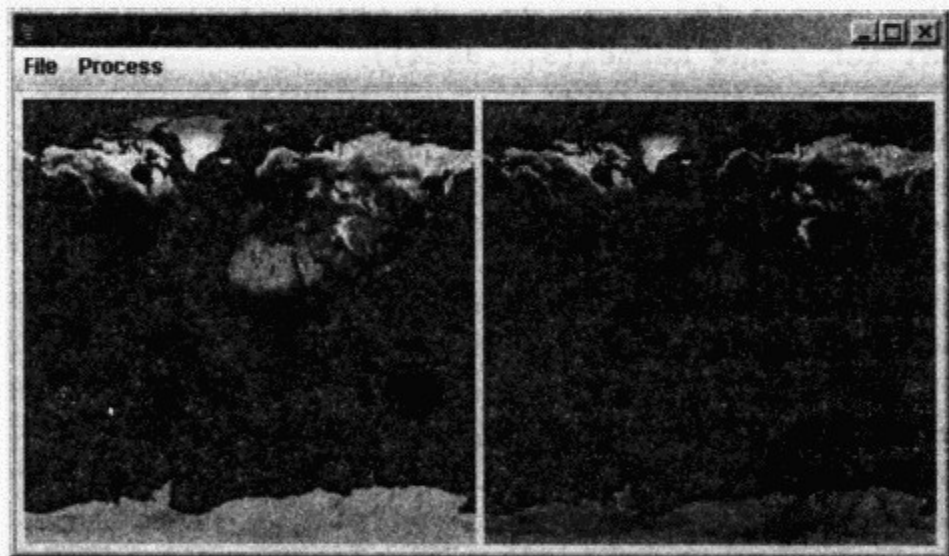


图4-5 图像锐化是这个图像处理示例程序所支持的一种操作

这个程序是图像处理系统的一个例子，它提供了图像I/O、运算和显示等基本功能。程序的主框架包含2个菜单，“File”菜单包含如下选项：打开一个图像文件；使用AWT工具打开一个图像文件；保存一幅图像到文件；退出程序。“Process”菜单则可以选择不同的图像处理操作。主框架的内容面板包含2个ImagePanel对象，源图像显示在左边，处理过的图像则显示在右边。

ImagePanel类继承于JPanel类，它显示一幅BufferedImage图像，图像可以通过构造函数或者setImage方法传递给ImagePanel对象。

程序实现了两种不同的方法来载入图像文件。一种方法使用ImageIO类的read方法来将图像直接载入BufferedImage对象中（第82行），另一种方法使用AWT方式的图像载入功能，并以绘制的方式获得BufferedImage对象（第92~101行）。一个JFileChooser对象被用来让用户选择要打开的图像文件。

程序实现了一些操作来执行常用的图像处理任务，包括：平滑、锐化、边缘检测、重缩放、旋转及灰度转换。平滑算子是用如下的 3×3 核定义的一个卷积：

$$\begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

114

锐化算子使用如下核进行定义：

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

边缘检测算子使用如下的核进行定义：

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

重缩放操作使用RescaleOp类，它用于增加图像的亮度。

旋转操作使用了AffineTransformOp类，通过一个AffineTransform对象定义了一个 $\pi/6$ 的旋转，然后将它应用于AffineTransformOp对象。

灰度转换操作使用ColorConvertOp类，来将源图像转化成一幅灰度图像。

复制操作简单地将处理过的图像拷回原来的图像中，以便继续应用其他处理。

处理过的图像可以保存为一个PNG格式的文件，用一个JFileChooser对象来选择要写入的文件，ImageIO的静态方法save用来保存一幅图像。

4.5 创建分形图像

在上一节里，讨论了如何创建、载入、保存、处理和显示一个BufferedImage，还可以通过执行底层操作，直接操纵BufferedImage中的像素。

Raster类封装了BufferedImage的像素数据。WritableRaster是Raster的子类，它是可写入的。要从一个BufferedImage中得到一个WritableRaster对象，可以使用getRaster方法：

```
BufferedImage bi = new BufferedImage(640,480,BufferedImage.TYPE_ARGB);
WritableRaster raster = bi.getRaster();
```

Raster类提供了一些方法来获取像素数据，WritableRaster用来添加设置像素数据的方法。

```
int[] getPixel(int x,int y,int[] data); //以数组形式返回指定像素的采样
float[] getPixel(int x,int y,float[] data);
double[] getPixel(int x,int y,double[] data);
int[] getPixels(int x,int y,int w,int h,int[] data); //返回一个包含像素矩形所有采样的数组，每个数组元素对应一个采样
float[] getPixels(int x,int y, int w,int h,float[] data);
double[] getPixels(int x,int y, int w,int h,double[] data);
void setPixel(int x,int y,int[] data); //使用输入样本的数组设置指定像素
void setPixel(int x,int y,float[] data);
void setPixel(int x,int y,double[] data);
```



```
void setPixels(int x,int y, int w,int h,int[] data);//使用输入样本数组设置指定像素矩形
void setPixels(int x,int y, int w,int h,float[] data);
void setPixels(int x,int y, int w,int h,double[] data);
```

参数 x 、 y 定义了像素的位置， w 、 h 定义了像素集合矩阵的尺度。 $data$ 数组保存像素数据，数组的大小取决于图像的类型。例如，如果BufferedImage是TYPE_INT_RGB的，那么每一个像素的数据数组就包含三个元素，分别标识RGB的三个分量的数值。

通过WritableRaster对象，图像的内容可以在像素层面上创建起来，逐个像素的图像绘制的方法可以通过建立一个分形图像的例子来说明。分形是一个自相似的几何结构，分形通常表现出很大的复杂度，尽管它们可能只是通过一些相当简单的程序绘制的。Mandelbrot集(Mandelbrot set)是一个著名的分形例子，它定义在复平面上。一个复数的形式如下：

$$x + iy$$

此处 x 、 y 都是实数， i 满足 $i^2 = -1$ 。两个复数 $z_1 = x_1 + iy_1$ 和 $z_2 = x_2 + iy_2$ 的加法和乘法定义如下：

$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$$

$$z_1 z_2 = (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + y_1 x_2)$$

复数的绝对值为 $|x + iy| = \sqrt{x^2 + y^2}$ 。如果将 (x, y) 的值看做点的坐标，一个复数就可以看做2D平面上的一个点，复数的绝对值对应该点到原点的距离。

为了定义Mandelbrot集，考虑在复平面上的迭代：

$$z_{n+1} = z_n^2 + c$$

c 是一个复数，迭代的起始点为 $z_0 = 0$ 。给定一个 c ，该迭代会产生一个复数序列： $z_0, z_1, \dots, z_n, \dots$ 。可以知道，这个序列或者趋向于无穷，或者保持有界。Mandelbrot集定义成使得该序列有界的所有点 c 的集合。Mandelbrot集复杂得惊人，它包含递归的自相似子结构。已知的是，Mandelbrot集位于以原点为圆心、半径为2的圆内。同时，在迭代的任何一步，如果点 c 超出了该圆的范围，该序列将会趋向于无穷，同时对应的点 c 就不再属于Mandelbrot集了。

程序清单4-5实现了基于Mandelbrot集来创建一幅图像的方法。这个例子创建了一个近似Mandelbrot集的图像。迭代过程针对每一个像素展开，同时，迭代的次数用于对颜色进行编码从而建立起图像。运行示例如图4-6所示。

程序清单4-5 Mandelbrot.java

```
1 package chapter4;
2
3 import javax.swing.*;
4 import java.awt.*;
5 import java.awt.image.*;
6 //定义Mandelbrot类，继承自JApplet类，演示使用Mandelbrot集创建图像
7 public class Mandelbrot extends JApplet {
8     public static void main(String s[]) {
9         JFrame frame = new JFrame();//创建主窗口
10        frame.setTitle("Mandelbrot set");//设置标题栏
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
12        JApplet applet = new Mandelbrot();//创建Mandelbrot对象
13        applet.init();//初始化
14        frame.getContentPane().add(applet);//添加applet到frame内容窗格
15        frame.pack();
16        frame.setVisible(true);//设置窗口可见
17    }
18    //重写初始化方法
```

```
19 public void init() {
20     JPanel panel = new MandelbrotPanel(); //创建MandelbrotPanel对象
21     getContentPane().add(panel); //加入内容窗格
22 }
23 }
24 //定义MandelbrotPanel类，继承自JPanel类
25 class MandelbrotPanel extends JPanel{
26     BufferedImage bi;
27
28     public MandelbrotPanel() {
29         int w = 500; //x轴的分段，即x轴方向501个点
30         int h = 500; //y轴的分段，即y轴方向501个点，总共检测501*501个点
31         setPreferredSize(new Dimension(w, h)); //面板大小500*500像素
32         setBackground(Color.white);
33         bi = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB); //新建图像bi
34         WritableRaster raster = bi.getRaster();
35         int[] rgb = new int[3]; //rgb数组用来储存某一点的RGB值
36         float xmin = -2; //初始点 (-2, -2)
37         float ymin = -2;
38         float xscale = 4f/w; //相邻点之间的间隔
39         float yscale = 4f/h;
40         for (int i = 0; i < h; i++) {
41             for (int j = 0; j < w; j++) {
42                 float cr = xmin + j * xscale;
43                 float ci = ymin + i * yscale;
44                 int count = iterCount(cr, ci);
45                 rgb[0] = (count & 0x07) << 5; //返回值的最后三位*32作为R值
46                 rgb[1] = ((count >> 3) & 0x07) << 5; //返回值的最后4-6位*32作为G值
47                 rgb[2] = ((count >> 6) & 0x07) << 5; //返回值的最后7-9位*32作为B值
48                 raster.setPixel(j, i, rgb); //设置该点(j,i)的RGB值
49             }
50         }
51     }
52     //迭代算法
53     private int iterCount(float cr, float ci) {
54         int max = 512;
55         float zr = 0;
56         float zi = 0;
57         float lengthsq = 0; //到原点的距离的平方
58         int count = 0;
59         while ((lengthsq < 4.0) && (count < max)) {
60             float temp = zr * zr - zi * zi + cr;
61             zi = 2 * zr * zi + ci;
62             zr = temp;
63             lengthsq = zr * zr + zi * zi; //计算距离的平方
64             count++;
65         }
66         return max-count;
67     }
68     //重写组件绘制方法
69     public void paintComponent(Graphics g) {
70         super.paintComponent(g);
71         g.drawImage(bi, 0, 0, this);
72     }
73 }
```


这个程序在如下方形区域内，建立了一个在复平面上描述Mandelbrot集的图像：

$$-2 \leq x, y \leq 2$$

程序创建一个大小为 500×500 的整型RGB像素类型的BufferedImage对象。从图像对象中得到了一个WritableRaster对象（第34行）。图像的像素通过该对象进行设置，像素的索引通过线性函数映射到复平面的坐标。

迭代通过方法iterCount执行。每个复数用两个实数变量表示，如果迭代的值超过圆 $|z_n| > 2$ （或者等价地说 $|z_n|^2 > 4$ ），则迭代结束。每个像素最大的迭代次数限制为512，所以迭代的次数范围是 $[1, 512]$ 。计数器max-count用于对像素着色。它的范围是 $[0, 512]$ 。9位计数器的值，分成三个3位的RGB分量。

paintComponent方法把完整的图像画在MandelbrotPanel上，这是JPanel的一个子类。由于产生Mandelbrot图像需要大量的计算，所以图像可能需要过一段时间才显示在窗口上。

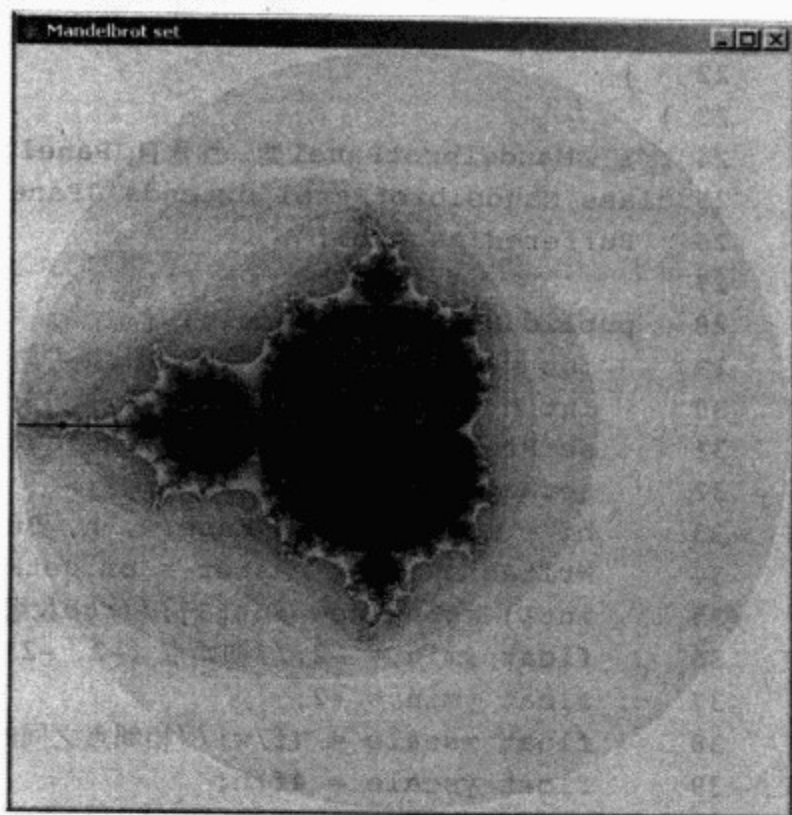


图4-6 根据迭代次数着色的Mandelbrot集

4.6 动画

动画为图形内容引入了动态变化，它通常建立起一个动态的视觉效果。动画产生一系列图像（帧），这些图像显示了场景中的变化。当这些帧以某一个速率被连续显示出来时（如每秒60帧），我们可以感受到场景中的连贯运动，而不是离散的图像。动画在图形模型中加入了时间维度，每一个时间点上的每一帧，本质上是一幅普通的静止图像。但是，帧的内容会随着时间的而改变，更高的帧速率（frame rate）表现出更平滑的动画，但是一个动画的帧速率受到图像绘制系统能力的限制。

118 在Java中，实现动画通常需要另外的线程来处理时间相关的变化。因为典型情况下，动画的运行是无限的，把动画的所有代码都放在事件分发线程中，会使GUI程序不能响应。但是，当使用Swing组件来显示图形动画的时候，应当避免从事件分发线程之外的其他线程直接操纵Swing组件，因为Swing组件并不是线程安全的（thread safe）。例如，在动画线程中，不能使用getGraphics()方法及使用Graphics对象来描绘图形。有几个Swing组件的方法是可以从其他的线程中安全地调用的：

```
public void repaint()
public void revalidate()
```

因此，在Swing组件上创建动画的合适方法是，将图像绘制与模型变化分离开来。绘制图形的代码只放在Swing组件的paintComponent方法中，而动画逻辑则放在不含图形代码的独立线程中。当一帧图像的数据准备好之后，调用repaint()方法触发图形绘制。一个典型的多线程动画的框架如下所示：

```
public void paintComponent(Graphics g){
    <*>render a frame*> //具体某一帧的绘制
}
public void run(){
```



```

    while(true){
        <*update from data*>//图像数据的更新
        repaint;
        try{
            Thread.sleep(sleepTime);
        }catch(InterruptedException ex){}
    }
}

```

paintComponent方法包含了生成一帧图像的所有代码。*Runnable*接口（Runnable interface）或者*Thread*类（Thread class）的run方法，经过重写用来实现动画。通常，它包含一个持续地产生帧的无限循环，每一帧通过调用repaint方法来产生。在两帧之间，线程总是休眠一段特定的时间。Thread类的sleep方法以毫秒为单位设定一段休眠时间。

程序清单4-6创建了一个模拟下雨的简单场景。许多垂直的线段正在向下移动。线段的位置和长度是随机的。程序的运行示例如图4-7所示。

程序清单4-6 Rain.java

```

1 package chapter4;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.event.*;
6 import java.util.*;
7 import javax.swing.*;
8 //定义Rain类，继承自JApplet类，演示下雨动画
9 public class Rain extends JApplet {
10     public static void main(String s[]) {
11         JFrame frame = new JFrame();//创建主窗口
12         frame.setTitle("Rain");//设置标题栏
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
14         JApplet applet = new Rain();//创建Rain对象
15         applet.init();//初始化
16         frame.getContentPane().add(applet);//添加applet到frame内容窗格
17         frame.pack();
18         frame.setVisible(true); //设置窗口可见
19     }
20     //重写初始化方法
21     public void init() {
22         JPanel panel = new JPanel();//创建RainPanel对象
23         getContentPane().add(panel);//加入内容窗格
24     }
25 }
26 //定义RainPanel，继承自JPanel，实现线程Runnable接口
27 class RainPanel extends JPanel implements Runnable{
28     Point2D.Double[] pts = new Point2D.Double[1200];//储存每条线段的起始位置
29
30     public RainPanel() {
31         setPreferredSize(new Dimension(640, 480));//设置组件首选大小
32         setBackground(Color.gray);//设置背景色
33         for (int i = 0; i < pts.length; i++) {
34             pts[i] = new Point2D.Double(Math.random(), Math.random());//随机生成点数组
35         }
36         Thread thread = new Thread(this);//新建线程
37         thread.start();//线程启动

```

119


```

38 }
39 //重写组件绘制方法
40 public void paintComponent(Graphics g) {
41     super.paintComponent(g);
42     g.setColor(Color.white);
43     for (int i = 0; i < pts.length; i++) {
44         int x = (int)(640*pts[i].x);//换算为屏幕上的位置
45         int y = (int)(480*pts[i].y);
46         int h = (int)(25*Math.random());//设置随机长度
47         g.drawLine(x, y, x, y+h);//绘制表示雨线的线段
48     }
49 }
50 //线程运行方法
51 public void run() {
52     while(true) {
53         for (int i = 0; i < pts.length; i++) {
54             double x = pts[i].getX();//取得该点当前位置
55             double y = pts[i].getY();
56             y += 0.1*Math.random();//向下移动0-1/10屏幕高度的距离
57             if (y > 1) { //超出屏幕最下方
58                 y = 0.3*Math.random();//在0-3/10个屏幕高度内随机位置
59                 x = Math.random();
60             }
61             pts[i].setLocation(x, y);//设定新位置
62         }
63         repaint();//重绘
64         try {
65             Thread.sleep(100);//休眠100ms
66         } catch (InterruptedException ex) {}
67     }
68 }
69 }

```

120

RainPanel类继承了JPanel类，同时实现了Runnable接口。一个Point2D.Double数组pts用来储存线段的位置（第28行），它们通过随机的值进行初始化。

paintComponent方法基于pts数组的值（根据组件的大小进行缩放）和随机的长度，来绘制垂直的线段。

在Rain的构造函数中创建了一个新的线程，用于执行由该类的run方法所提供的代码。run方法（第51行）包含一个无限循环，以更新保存在pts数组中的线段位置。每一个点都在y轴方向上增加一个随机的量，使得线段“向下掉”。当一条线段到达底部后，就把该点重置到顶部区域的随机位置。每当数组更新之后，都调用repaint方法来重画面板。然后，线程休眠100ms。

RainPanel的一个实例放在小程序Rain中，它包含了一个标准的main方法，用来像应用程序一样运行这个动画。

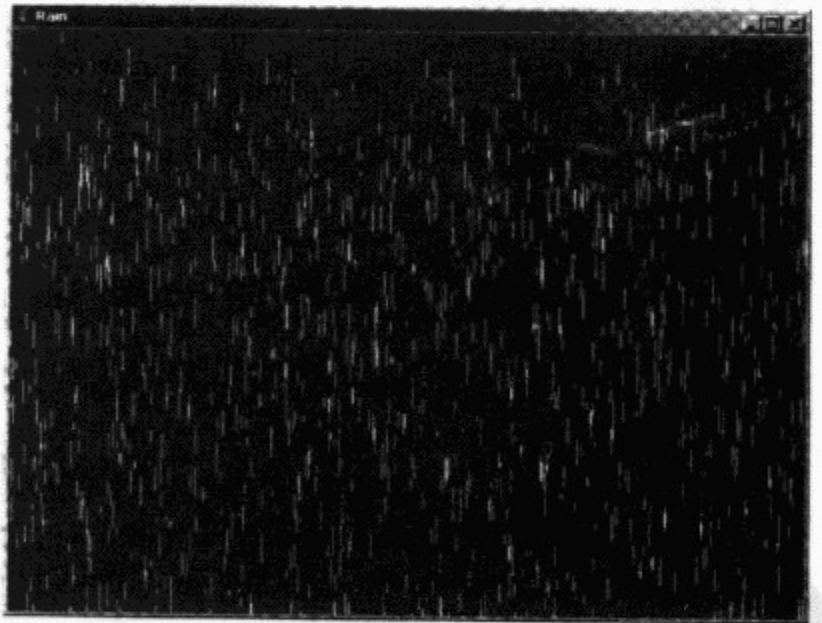


图4-7 一个下雨的动画

除了创建自己的线程，还可以使用Swing的Timer类。一个Timer对象以预定义的速率周期地产生一个动作事件，该事件会触发监听器，监听器可以执行帧的绘制。要设立一个Timer对象，可以在构造函数中为它指定周期值和监听器，然后调用它的start()方法：

```
Timer timer = new Timer(period, listener);
timer.start();
```

ActionListener对象应该在actionPerformed方法中实现图像绘制：

```
public void actionPerformed(ActionEvent event){
    <*do frame rendering*>
}
```

对于动画，Timer类提供了比直接创建线程更为方便的方法。由于actionPerformed方法在事件分发线程中进行调用，所以在该方法中执行Swing组件上的图像绘制是安全的。

121

程序清单4-7演示了Timer类的应用。一个实时模拟时钟如图4-8所示，时钟的时间来自于系统时间，时钟持续更新，以达到时钟运动的视觉效果。

程序清单4-7 Clock2D.java

```
1 package chapter4;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.event.*;
6 import java.util.Calendar;
7 import javax.swing.*;
8 //定义Clock2D类，继承自JApplet类，演示一个走动的时钟
9 public class Clock2D extends JApplet {
10     public static void main(String s[]) {
11         JFrame frame = new JFrame();//创建主窗口
12         frame.setTitle("Clock");//设置标题栏
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
14         JApplet applet = new Clock2D();//创建Clock2D对象
15         applet.init();//初始化
16         frame.getContentPane().add(applet);//添加applet到frame内容窗格
17         frame.pack();
18         frame.setVisible(true);//设置窗口可见
19     }
20     //重写初始化方法
21     public void init() {
22         JPanel panel = new ClockPanel();//创建ClockPanel对象
23         getContentPane().add(panel);//加入内容窗格
24     }
25 }
26 //定义ClockPanel类，继承自JPanel类，实现事件侦听接口ActionListener
27 class ClockPanel extends JPanel implements ActionListener{
28     AffineTransform rotH = new AffineTransform();//时针的变换
29     AffineTransform rotM = new AffineTransform();//分针的变换
30     AffineTransform rotS = new AffineTransform();//秒针的变换
31
32     public ClockPanel() {
33         setPreferredSize(new Dimension(640, 480));//设置组件首选大小
34         setBackground(Color.white);//设置组件背景颜色
35         Timer timer = new Timer(500, this);//建立500ms周期的Timer
36         timer.start();//timer启动
```



```

37 }
38 //重写组件绘制方法
39 public void paintComponent(Graphics g) {
40     super.paintComponent(g);
41     Graphics2D g2 = (Graphics2D)g;
42     g2.translate(320,240); //将 (320, 240) 设为坐标原点0
43     //创建钟面
44     Paint paint = new GradientPaint
45     (-150,-150,Color.white,150,150,Color.gray);
46     g2.setPaint(paint);
47     g2.fillOval(-190, -190, 380, 380); //填充圆
48     g2.setColor(Color.gray);
49     g2.drawString("Java 2D", -20, 80); //绘制字符串"Java 2D"
50     Stroke stroke = new BasicStroke(3);
51     g2.setStroke(stroke);
52     g2.drawOval(-190, -190, 380, 380); //画钟面的轮廓(圆)
53     for (int i = 0; i < 12; i++) { //tick marks
54         g2.rotate(2*Math.PI/12); //坐标旋转30度
55         g2.fill3DRect(-3, -180, 6, 30, true); //填充矩形
56     }
57     //创建指针
58     Shape hour = new Line2D.Double(0, 0, 0, -80);
59     hour = rotH.createTransformedShape(hour); //按rotH变换过之后传给hour
60     Shape minute = new Line2D.Double(0, 0, 0, -120);
61     minute = rotM.createTransformedShape(minute); //按rotM变换过之后传给minute
62     Shape second = new Line2D.Double(0, 0, 0, -120);
63     second = rotS.createTransformedShape(second); //按rotS变换过之后传给second
64     g2.setColor(Color.black);
65     g2.setStroke(new BasicStroke(5,
66     BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND)); //宽度为5
67     g2.draw(hour);
68     g2.draw(minute);
69     g2.setStroke(new BasicStroke(2)); //宽度为2
70     g2.draw(second);
71 }
72 //Timer定时触发事件, 执行此方法
73 public void actionPerformed(ActionEvent e) {
74     int hour = Calendar.getInstance().get(Calendar.HOUR);
75     int min = Calendar.getInstance().get(Calendar.MINUTE);
76     int sec = Calendar.getInstance().get(Calendar.SECOND);
77     rotH.setToRotation(Math.PI * (hour+min/60.0)/6.0); //设定旋转的角度
78     rotM.setToRotation(Math.PI * min /30.0);
79     rotS.setToRotation(Math.PI * sec /30.0);
80     repaint(); //重绘
81 }
82 }

```

ClockPanel类继承了JPanel, 实现了ActionListener接口。三个AffineTransform字段定义了时针、分针和秒针的旋转(第28~30行)。

paintComponent方法描绘钟面和三根指针。钟面包括一个以灰色渐变涂色方式填充的圆, 十二个用填充的3D矩形创建的刻度线, 以及一个字符串“Java 2D”。时针、分针和秒针用线段画出, 它们的位置由旋转字段决定, 旋转字段定义了当前时间下正确的旋转角度。程序将旋转变换应用于对应指针, 这是从12点钟的位置开始执行的。

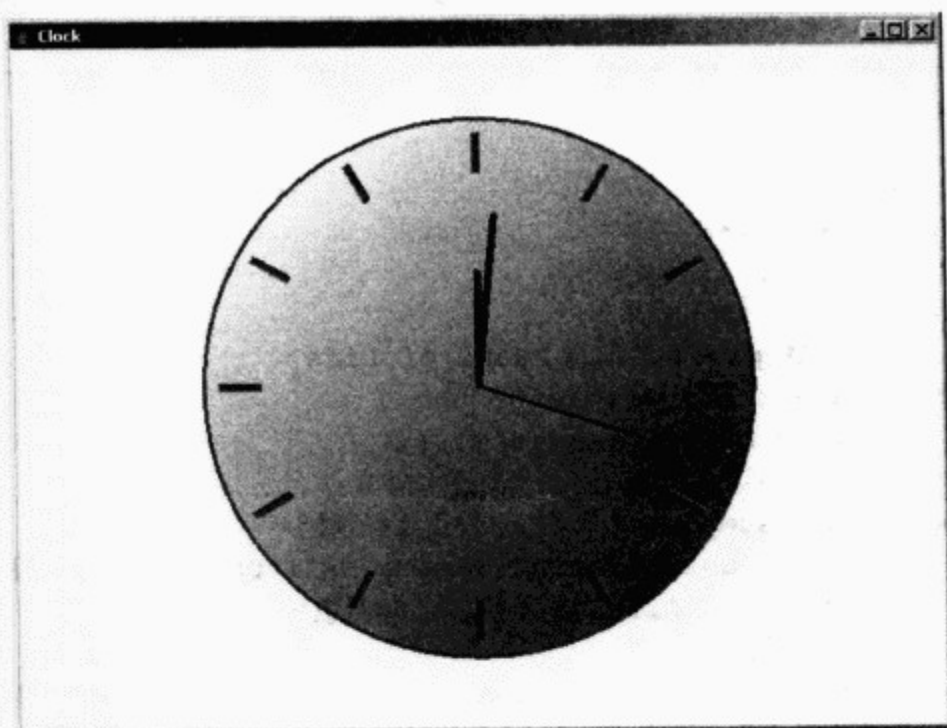


图4-8 一个实时模拟时钟

一个Timer对象在ClockPanel的构造函数中进行创建和运行。它将ClockPanel对象作为监听器，设置了500ms的周期（第35行）。为了响应Timer产生的动作事件，actionPerformed方法实现了动画函数，它使用Calendar类来获取当前系统的时间，然后设置三个合适的旋转角度。在旋转更新之后，调用repaint方法来更新显示。

细胞自动机（cellular automaton）是一个简单的网格迭代系统，该系统基于一个固定的规则集进行进化。许多细胞自动机可以产生令人惊讶的复杂图案，2D细胞自动机定义在一个2D网格上。每一个细胞有两个状态：黑和白（也称为生和死）。一个细胞有八个邻居。在某些系统中，只考虑相邻的四个细胞。系统的迭代基于系统的上一个状态为每一个细胞赋予下一个状态。每一个细胞遵循相同的规则集，且新的状态仅仅取决于该细胞和它邻居们当前的状态。例如，图4-9显示了一次迭代，其规则是：“如果细胞在当前格局下只有一个邻居是黑的，那么该细胞就是黑的。”

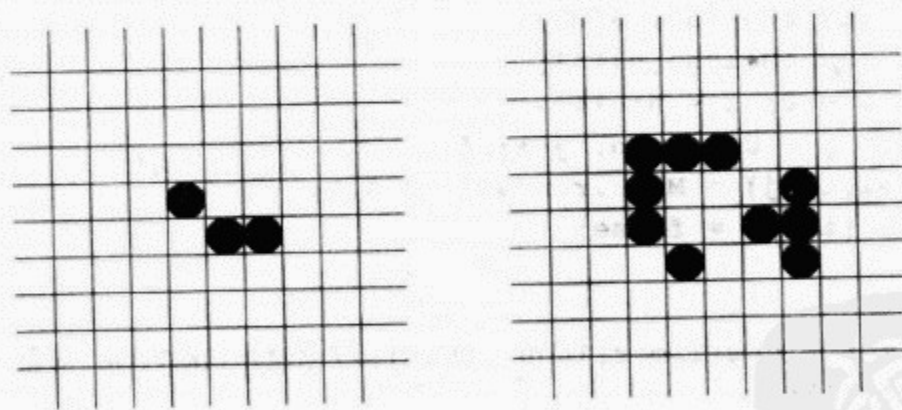


图4-9 细胞自动机进化的一步

细胞自动机的一个著名的例子是Conway的生命游戏（Game of Life），即程序清单4-8。程序的运行示例如图4-10所示。

生命游戏是一个2D的细胞自动机，其规则相当简单：

- 1.（出生）。如果一个死亡的细胞有三个活着的邻居，它就变成活的。
- 2.（生存）。如果一个活着的细胞有二个或者三个活的邻居，它就仍然活着。
- 3.（死亡）。在其他情况下，该细胞死亡。

程序清单4-8 Life.java

```
1 package chapter4;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import java.awt.geom.*;
7 //定义Life类, 继承自JApplet类, 演示了Game of Life
8 public class Life extends JApplet {
9     public static void main(String s[]) {
10         JFrame frame = new JFrame();//创建主窗口
11         frame.setTitle("Game of Life");//设置标题栏
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
13         JApplet applet = new Life();//创建Life对象
14         applet.init();//初始化
15         frame.getContentPane().add(applet);//添加applet到frame内容窗格
16         frame.pack();
17         frame.setVisible(true);//设置窗口可见
18     }
19     //重写初始化方法
20     public void init() {
21         JPanel panel = new LifePanel();//创建LifePanel对象
22         getContentPane().add(panel);//加入内容窗格
23     }
24 }
25 //定义LifePanel类
26 class LifePanel extends JPanel implements ActionListener{
27     int n = 30;//30*30
28     boolean[][] cells1;//储存当前配置信息
29     boolean[][] cells2;//储存下一次配置信息
30
31     public LifePanel() {
32         setPreferredSize(new Dimension(400, 400));
33         setBackground(Color.white);
34         cells1 = new boolean[n][n];
35         cells2 = new boolean[n][n];
36         for (int i = 0; i < n; i++) {
37             for (int j = 0; j < n; j++) {
38                 cells1[i][j] = Math.random() < 0.1;
39                 cells2[i][j] = false;
40             }
41         }
42         Timer timer = new Timer(1000, this);//创建并启动Timer对象
43         timer.start();
44     }
45     //重写组件绘制方法
46     public void paintComponent(Graphics g) {
47         super.paintComponent(g);
48         Graphics2D g2 = (Graphics2D)g;
49
50         g2.setColor(Color.lightGray);//网格线的颜色
51         int p = 0;
52         int c = 16;//两条线之间的间隔
53         int len = c*n;
54         for (int i = 0; i <= n; i++) { //绘制网格
```

```

55     g2.drawLine(0, p, len, p);
56     g2.drawLine(p, 0, p, len);
57     p += c;
58 }
59 g2.setColor(Color.black);
60 for (int i = 0; i < n; i++) {
61     for (int j = 0; j < n; j++) {
62         if (cells1[i][j]) { //如果当前细胞存活
63             int x = i*c;
64             int y = j*c;
65             g2.fillOval(x, y, c, c); //画黑色实心圆圈
66         }
67     }
68 }
69 }
70 //事件响应函数
71 public void actionPerformed(ActionEvent e) {
72     boolean[][] cells = cells1;
73     for (int i = 0; i < n; i++) {
74         for (int j = 0; j < n; j++) {
75             cells2[i][j] = cells[i][j];
76             int nb = neighbors(cells, i, j); //获取当前细胞近邻存活的个数
77             if (nb == 3)
78                 cells2[i][j] = true; //近邻存活3个，自己存活
79             if (nb < 2 || nb > 3)
80                 cells2[i][j] = false; //不等于2、3，该细胞死亡
81             //近邻存活2个，维持自身情况
82         }
83     }
84     cells1 = cells2; //更新当前配置信息
85     cells2 = cells;
86     repaint(); //重绘
87 }
88 //获得当前细胞邻居的存活个数
89 private int neighbors(boolean[][] cells, int x, int y) {
90     int x1 = (x>0)?x-1:x;
91     int x2 = (x<n-1)?x+1:x;
92     int y1 = (y>0)?y-1:y;
93     int y2 = (y<n-1)?y+1:y;
94     int count = 0;
95     for (int i = x1; i <= x2; i++) {
96         for (int j = y1; j <= y2; j++) {
97             count += (cells[i][j])?1:0;
98         }
99     }
100     if (cells[x][y]) count--;
101     return count;
102 }
103 }

```

125

126

程序创建了一个 $n \times n$ 的面板来显示游戏，一个2D的boolean数组用于储存细胞的状态，初始状态是随机的，并且活细胞的概率为0.1。

一个1000ms的定时器用来推动游戏的进行（第42行）。LifePanel对象是定时器的行为监听器。每次事件触发，就执行游戏的下一步。为了避免反复地创建数组，所以只使用两个数组来存储细胞的配置（第28~29行）。cells1指向当前显示的细胞数组，cells2则为下一次配置。在面

板更新的计算结束之后，两个数组交换。

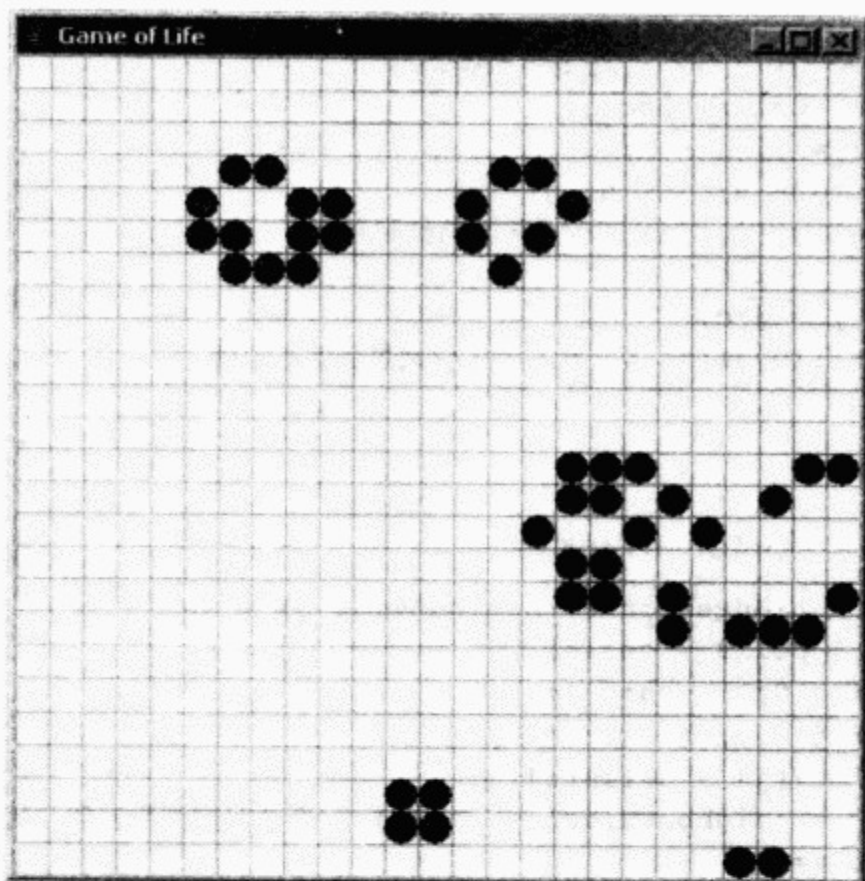


图4-10 生命游戏

4.7 打印

打印是计算机应用的一项普通任务。打印图形所涉及的绘制问题，本质上与在屏幕上的绘制问题是一样的。但是，打印确实存在一些特殊的问题，比如分页。Java API为打印提供了方便的工具。

PrinterJob类可以用来管理打印进程。下面的静态方法返回PrinterJob的一个实例：

```
static PrinterJob getPrinterJob();
```

在打印任务的开始，用户通常会用到一个关于打印机选项的对话框，可以简单地通过调用PrinterJob方法来做到这一点。

```
boolean printDialog();
```

如果用户选择进行打印，则该方法返回true，可以通过调用下面的方法来初始化打印：

```
void print() throws PrinterException;
```

打印的实际图形内容，可以使用Printable接口来定义。通过下面的函数，可以给一个PrinterJob对象选择一个实现了Printable接口的对象：

```
void setPrintable(Printable painter);
```

Printable接口提供了一个回调结构，用于定义打印任务的自定义绘制，该接口包括下列用于自定义实现的方法：

```
int print(Graphics g, PageFormat pf, int pageIndex);
```

这个方法的实现与paintComponent非常相似，在两个方法之间经常可以共享代码。Graphics参数可以转化为Graphics2D对象，该对象提供了使用2D图像绘制引擎的方法。所有的绘画、转化以及其他特征对于打印都是适用的。

pageIndex参数提供了当前绘制页的页码，页面从0开始计数。对于该方法的实现，如果这

一页不需要打印，则返回NO_SUCH_PAGE，如果打印了这一页，则返回PAGE_EXISTS。

PageFormat对象包含了打印机的页面设置信息，下面是一些PageFormat类的方法：

```
int getOrientation();
double getWidth();
double getHeight();
double getImageableX();
double getImageableY();
double getImageableWidth();
double getImageableHeight();
```

127

最后四个方法用于获得该页可打印区域的边界矩形。

程序清单4-9演示了这些类的用法，在Java应用中实现了打印功能。程序显示了带有文字“Welcome”和标签“Print”面板的窗口。如果点击按钮，会显示一个打印机选项对话框，选定打印机以后，将会在两个页面上打印相同的文本字符串，如图4-11所示。绘制图像将对照可打印区域进行裁剪，但各页面的打印内容要对齐。

程序清单4-9 Printing.java

```
1 package chapter4;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.event.*;
6 import java.awt.print.*;
7 import javax.swing.*;
8 //定义PrintGraphics类，继承自JFrame类，实现事件侦听接口
9 public class PrintGraphics extends JFrame implements ActionListener {
10     public static void main(String[] args) {
11         JFrame frame = new PrintGraphics();//创建主窗口
12         frame.setTitle("Printing");//设置标题栏
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭处理
14         frame.pack();
15         frame.setVisible(true);//设置窗口可见
16     }
17
18     PrinterJob pj;
19     PrintPanel painter;
20     //按钮事件响应处理函数
21     public void actionPerformed(ActionEvent e) {
22         if (pj.printDialog()) {
23             try {
24                 pj.print();//打印
25             } catch (PrinterException ex) {
26                 ex.printStackTrace();
27             }
28         }
29     }
30
31     public PrintGraphics() {
32         Container cp = this.getContentPane();
33         cp.setLayout(new BorderLayout());
34         JButton button = new JButton("Print");//添加按钮和事件侦听器
35         cp.add(button, BorderLayout.SOUTH);
36         button.addActionListener(this);
```



```

37     painter = new JPanel();
38     cp.add(painter, BorderLayout.CENTER);
39     pj = PrinterJob.getPrinterJob();
40     pj.setPrintable(painter);
41 }
42 }
128 43 //定义PrintPanel类
44 class PrintPanel extends JPanel implements Printable {
45     public PrintPanel() {
46         setPreferredSize(new Dimension(800, 400)); //设置组件首选大小
47         setBackground(Color.white); //设置背景颜色
48     }
49     //print方法
50     public int print(Graphics g, PageFormat pf, int pageIndex) {
51         switch (pageIndex) {
52             case 0:
53                 draw(g);
54                 break;
55             case 1:
56                 g.translate(-(int)pf.getImageableWidth(), 0); //调整坐标系, 新原点为(-width,0)
57                 draw(g);
58                 break;
59             default:
60                 return NO_SUCH_PAGE;
61         }
62         return PAGE_EXISTS;
63     }
64     //重写组件绘制方法
65     public void paintComponent(Graphics g) {
66         super.paintComponent(g);
67         draw(g);
68     }
69     //绘制函数
70     private void draw(Graphics g) {
71         g.setFont(new Font("Serif", Font.BOLD, 144)); //设置字体
72         g.drawString("Welcome!", 200, 300); //绘制字符串
73     }
129 74 }

```

PrintPanel类继承了JPanel, 实现了Printable接口。程序的主窗口加入了PrintPanel的一个实例, 它同样也用于打印。该面板使用一种144点的字体显示字符串“Welcome”, 所定义的私有方法draw用于执行画图操作(第70行), paintComponent和print方法调用draw方法。

print方法还包括了处理分页的逻辑。第一页(pageIndex=0)直接进行打印, 在打印第二页时, 执行了一个平移变换来移动该页, 使得它覆盖第一页的右侧区域。为了确定精确的偏移量, 我们使用PageFormat对象来获取可打印区域的宽度(第56行)。



图4-11 多页面打印

主要的类和方法

- `javax.awt.geom.GeneralPath.curveTo(...)` 用于构建3次曲线段的方法。
- `javax.awt.image.BufferedImage` 封装了一幅图像的类。
- `javax.awt.image.BufferedImageOp` 图像处理操作的接口。
- `java.swing.Timer` 周期性产生动作事件的类。
- `java.lang.Runnable` 用于定义作为单独线程执行的代码的接口。
- `java.lang.Thread` 封装了程序执行的线程的类。
- `java.lang.Thread.sleep(long ms)` 使线程休眠一段特定时间的方法。
- `java.util.Calendar` 封装了日期和时间日历的类。
- `java.awt.image.Raster` 封装图像数据的类。
- `java.awt.image.WritableRaster` 封装可改写的图像数据的类。
- `java.awt.print.PrinterJob` 打印管理的类。
- `java.awt.print.Printable` 定义打印内容的接口。

关键术语

- **B样条曲线** (B-spline curve) 一种用分段多项式参数方程和一组控制点混合定义的曲线。
- **Bézier曲线** (Bézier curve) 一种用多项式参数方程和一组控制点混合定义的曲线。
- **NURBS** 非均匀有理B样条曲线。
- **图像处理** (image processing) 数字图像的计算机操作，用来增强图像或者提取信息。
- **卷积** (convolution) 一类线性操作，通常用在信号和图像处理中。
- **核** (kernel) 用来定义卷积的一个函数。
- **复数** (complex numbers) 对实数的一种扩展。
- **复平面** (complex plane) 把复数集看做是一个平面上的点的集合。
- **帧速率** (frame rate) 用帧/每秒 (fps) 来衡量的动画速率。
- **线程** (thread) 运行程序中的一个执行分支。在一个多线程的环境中，一个程序可能有多个线程同时运行。
- **细胞自动机** (cellular automata) 细胞网格上的动态系统，基于一个简单的规则进行进化，该规则集根据每个细胞自身和邻居的上一个状态，来决定此细胞的下一个状态。

本章提要

- 本章介绍了2D计算机图形学的一些高级技术。
- B样条曲线是2D几何的重要建模工具，一条B样条曲线能够被转化为一组Bézier曲线，而Bézier曲线可以由Graphics2D对象直接绘制，每一条Bézier曲线的控制点都是B样条曲线控制点的线性组合。
- Java 2D提供了一组基本的图形基元，我们讨论了如何通过实现Shape接口来构建自定义的基元。尽管不能继承GeneralPath类，但仍旧可在你自己的类里面包含它，从而利用它的实现。
- 本章简单介绍了图像处理的相关内容。Java 2D对图像的读取、写入、绘制和处理提供了支持，BufferedImage类是Java 2D中表示图像的主要方式。图像处理操作实现了BufferedImageOp接口，提供了处理BufferedImage对象的简便方法。
- 在程序中，图像同样可以在像素的层面上进行创建和操纵，Raster和WritableRaster类提供了访问BufferedImage对象中的像素数据的方法。此外，还给出了一个构建Mandelbrot集合图像的例子。
- 动画是计算机图形学的一个重要部分。在Java中实现一个2D动画需要多线程的支持，以便把持续的图像绘制与一般的用户界面处理分离开来。Thread类和Runnable接口提供了基本的多线程支持。Swing包的Timer类提供了触发周期性地绘制帧的便捷方法。

130

- Java的打印与Java 2D的一般图像绘制方案很一致。同样的Graphics2D类提供图形绘制引擎，Printable接口允许必要的回调结构来定义图形的输出，PrinterJob类使打印中的特殊任务变得容易。

复习题

- 4.1 给定一个如本章所定义的有 n 个控制点的B样条曲线，它可以转化成多少条Bézier曲线？
- 4.2 另一类B样条曲线对曲线的端点没有特殊的限制。第一段和最后一段与其他段一样处理。在每一段都使用相同的公式，用于将它转化为Bézier曲线。如此一来，生成的曲线不一定通过第一个和最后一个控制点。请给出这类B样条曲线的转化公式。
- 4.3 如果你对一幅图像先进行平滑操作，再进行锐化操作，你能恢复原图像吗？
- 4.4 对于 $c = i$ ，在Mandelbrot集合的定义中执行5次迭代。
- 4.5 对于 $c = 1$ ，在Mandelbrot集合的定义中执行5次迭代。
- 4.6 如果完成一帧的绘制要花费0.01秒，而且进程对于每一帧的休眠时间为50毫秒，计算帧速率。
- 4.7 对于有四个邻居的细胞自动机，总共有多少种不同的规则？

编程练习

- 4.1 实现复习题4.2中的那一类B样条曲线。
- 4.2 把Heart基元加入到程序2.3中，使它能够像其他形状一样，可以进行选择和绘制。
- 4.3 编写一个Triangle类，实现Shape接口。提供一个构造函数来定义三角形的三个顶点。
- 4.4 实现一个正 n 边形基元，定义该类以实现Shape接口，提供构造函数来指定边数 n 。
- 4.5 在程序4.4中添加一个取反操作。所有像素的颜色均转化成补色，即一种颜色的 r 、 g 、 b 分量变成 $1-r$ 、 $1-g$ 、 $1-b$ 。
- 4.6 编写一个Java程序，要有一个文本字段、一个按钮和一个文本区域，如图4-12所示。用户可以在文本域输入一个字符串，当按下“Print”键时，文本区域将会显示以“*”形成的字符串图案。

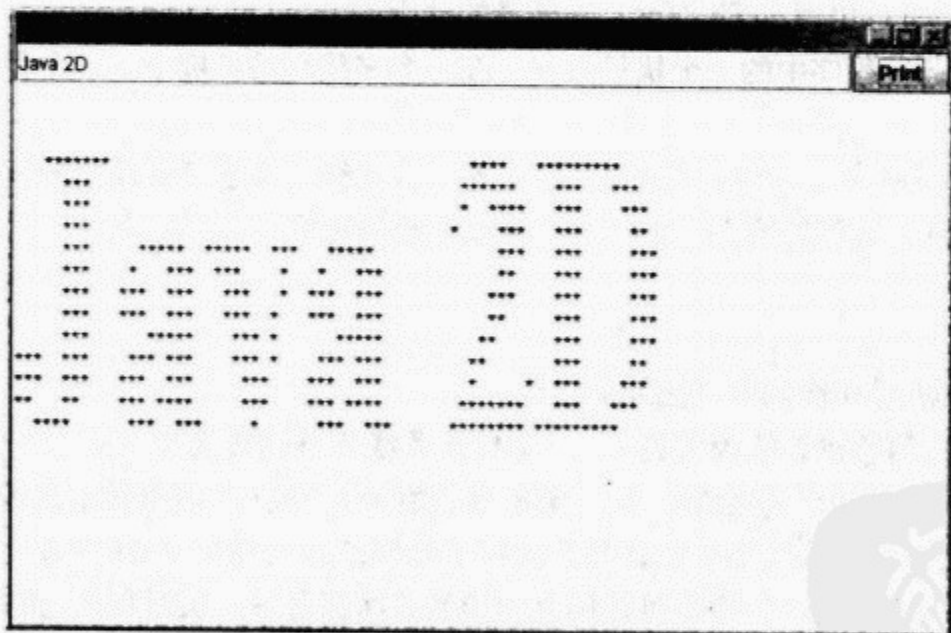


图4-12 ASCII艺术

- 4.7 修改程序4.5，使图像中可以看到复平面的一个矩形区域，用户可以通过拖动鼠标来选择矩形区域。
- 4.8 Julia集允许使用Mandelbrot集合相同类型的迭代。常数 c 总是固定的，迭代的起始点并不一定为0。如果在某点开始的迭代所产生的序列是有界的，则该点属于Julia集。当 $c = -0.672 + 0.435i$ 时，编写一个Java程序来显示Julia集。
- 4.9 修改程序清单3-6的程序，使其在一个支持 α 通道的离屏图像上根据合成规则进行绘制。图像绘制结束后，再把它绘制到显示屏上。

- 4.10 编写一个程序来显示一个有四片叶子的运转中的风扇。
- 4.11 编写一个Java 2D程序来显示在一个矩形内弹跳的一个球，如图4-13所示。初始化时球按照一条随机选定的路线运动，当球碰到矩形边界的时候，它以另一个随机的方向弹回。
- 4.12 编写一个Java程序，模拟一个指针显示的秒表。使用鼠标点击来操作秒表，秒表遵循3个状态的循环：开始—停止—重置。
- 4.13 按照以下规则实现一个细胞自动机，它基于每个细胞有四个邻居：
- 1) 如果一个白色细胞的黑色邻居数不为1，则它变成黑色。
 - 2) 如果一个黑色细胞的黑色邻居数为1或3，则它维持黑色。
 - 3) 其他情况下，细胞变成白色。
- 初始时屏幕中央有一个黑色细胞，一段时间后会出现如图4-14所示的令人惊讶的模式。使用一个Timer对象来处理动画进程，并允许用户通过点击鼠标来停止动画。

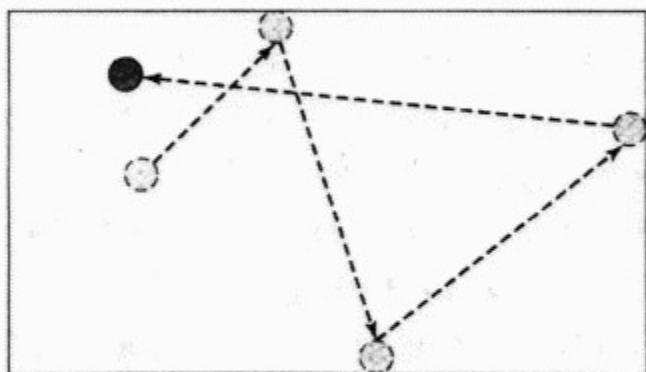


图4-13 跳跃的球

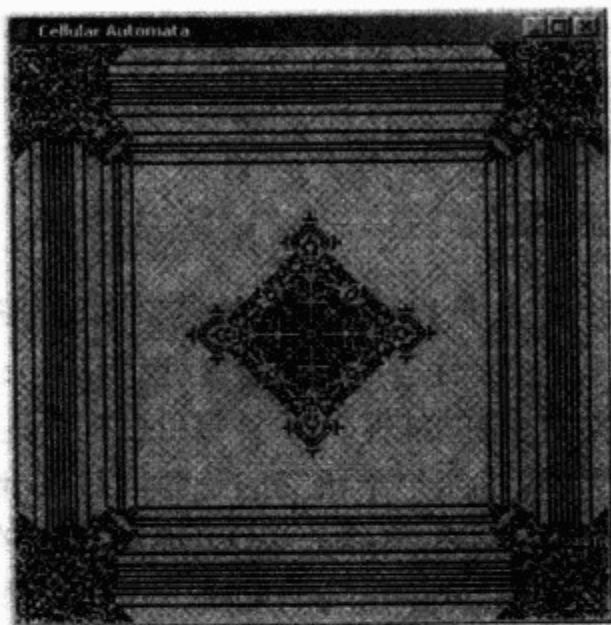


图4-14 一个简单的细胞自动机所产生的图案

- 4.14 编写一个Java程序来打印一个显示当前时间的模拟闹钟。
- 4.15 给程序清单2-3中的绘制程序添加一个“Print”菜单项。当选择该菜单项的时候，它会开始打印当前的绘制。

第5章 基本3D图形

学习目标

- 描述3D绘制过程。
- 概述Java 3D程序。
- 定义Java 3D场景图。
- 对场景图组件进行分类。
- 使用背景节点。
- 理解并使用作用范围边界。
- 修改活动场景图。

134
135

5.1 引言

虽然现实世界带给我们的感觉明显是3D的，但是我们通过双眼看见的视觉图像实际上却是2D的。在这背后是一种称为透视投影（perspective projections）的特殊映射机制，完成了从3D场景到2D图像的捕捉。3D计算机图形学的基本目标就是在计算机中模拟这一过程。

3D计算机图形学研究的是如何对3D世界进行建模和绘制。3D空间中的几何对象可能是零维的（点）、1D的（线）、2D（面）或3D的（实体）。物体可能有各种不同的材料属性。在虚拟的空间中，可能存在多个光源，每个光源具有不同的特征。捕捉虚拟世界场景的虚拟摄像机，可以放置在空间中的各个不同位置，每台摄像机可能有不同的特征。3D图形系统必须解决的问题，包括图形对象及其属性的表示，对图形对象辅助进行几何变换，所有图形对象组件的组织，以及整个场景的绘制等。

Java 3D是一个面向对象的3D计算机图形API。每个Java 3D程序的完整的图形模型，是作为一个称之为场景图（scene graph）的结构进行组织的。场景图中的每一个节点都是一个类的对象，用以表示众多图形实体中的一个。场景图结构提供了一个系统性的模型，使Java 3D绘制引擎能够自动地绘制由Java 3D程序所构造的场景。

本章将介绍3D图形系统的一些基本概念。特别地，你将学习到Java 3D的总体架构。同时，本章还对Java 3D场景图的概念和各种场景图组件进行了概要介绍。你将学会如何构造Java 3D场景图，并能编写简单的Java 3D程序。

5.2 3D绘制过程

绘制3D场景以生成图像（通常为2D），这本身就是一个复杂的过程。与2D图形不同的是，3D对象的绘制图像和它本身的3D版本大为不同，基于对象的绘制图像直接进行对象建模是不可行的。因此，3D图形系统需要构造一个虚拟世界，并在其中定义各种图形对象和光源。与2D图形中常见的现场构造场景不同，我们需要始终保留一个独立于绘制引擎的虚拟世界模型，在此虚拟世界中，以一种结构化的方式对场景进行观察，并由绘制引擎产生视图。图5-1简单演示了3D图形的一些概念。

静态图形场景的绘制过程，类似于用一台照相机拍摄一幅照片的过程。虚拟世界中包含虚

拟对象，这些对象可以反射各种光源发出的光。照相机放置在虚拟世界中的某个特定的点，并把虚拟世界中的可见部分沿着特定的方向投影到一个2D平面上。图形对象与视角都可以是动态变化的，因此，场景和根据场景绘制的图像都可能随时间发生变化。虚拟世界和现实世界之间还可能存在互动，用户和传感器的输入可能对虚拟世界的模型产生影响。

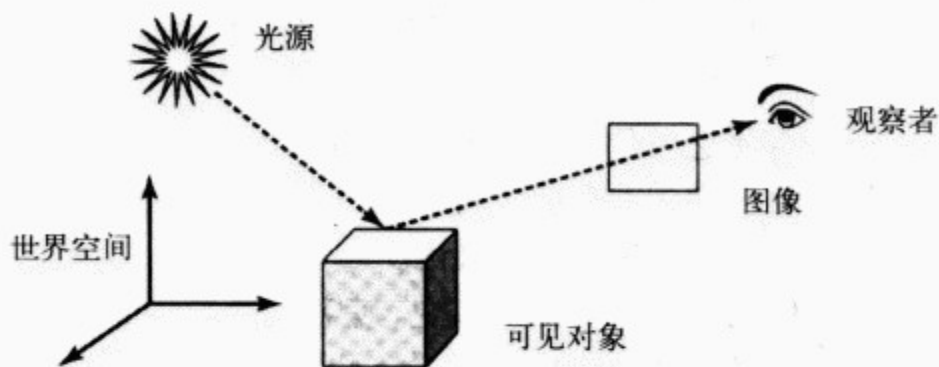


图5-1 3D图形的模型和视图

136

要实现或使用这样一个3D图形系统，必须考虑许多与虚拟世界的建模和场景绘制相关的问题，例如：

- 图形对象的几何属性。
- 图形对象的坐标和方向。
- 施加于图形对象和视角的几何变换。
- 图形对象的材料属性与纹理。
- 光源和它们的属性。
- 观察的投影类型。
- 视角位置、视域以及视角的其他属性。
- 光照和阴影模型。
- 不同组件的动态行为。
- 对用户输入的反应。

在3D图形系统中构造虚拟世界，最基本的问题是图形对象的几何描述。3D图形对象的基本组件包括点、线、面和实体。复杂对象则通常用简单的多边形网格来近似描述。3D图形系统通常会提供一些方便的机制，用于生成一些高级的几何体，例如3D文字和几何基元（例如球体、锥体与立方体等）。一些高级建模工具则还包含样条曲线和曲面。

几何变换是图形系统中的一项重要工具。我们通过几何变换把几何对象放置到虚拟世界的3D空间中，并在必要时改变它们的形状、大小和位置。3D仿射变换（3D affine transform）是虚拟世界空间中常用的一类变换，其他类型的几何变换，还包括更为一般的投影变换（projective transform）。投影变换是3D观察过程的重要组成部分。

除了几何属性，图形对象的外观属性决定了如何绘制此图形对象。外观属性包括颜色和纹理，以及用于更为复杂的明暗效果的材质属性。光照（Lighting、illumination 或 shading）策略决定了如何计算物体的颜色和亮度。光照模型的选择还会影响到绘制的结果。在某些光照模型下，一些特定的几何信息与物体的外观密切相关，例如物体表面各点的法线方向。物体表面上某点的法线方向是垂直于该点位置上切面的方向。例如在考虑镜面反射的Phong光照模型中，物体表面某点的亮度与观察方向和反射光方向之间的夹角有关。反射光向量则是由光照方向和物体表面的法线方向决定的。我们将在第9章详细介绍光照模型的细节。

3D视图变换过程总是涉及从3D场景到2D平面的投影转换。每个视图可以有很多参数来控

制其特征，投影转换可以是平行投影或透视投影。对一个特定的视图，虚拟世界中可见的区域通常是有限的，这一区域称为观察平截体（view frustum）。简单的数学意义上的投影变换，未必足以完成绘制任务。对象之间的相对位置可能会对绘制过程产生重要影响，例如，对象的一部分可能被另一个对象遮挡。只有在恰当地解决这些问题之后，才能得到可接受的视觉效果。

当然，3D绘制并不仅限于静态场景。虚拟世界可以随时间变化。观察系统可以与一个动态装置关联，例如和一个头戴式显示装置关联。绘制过程的动态效果可能包括动画或交互。交互（interaction）是由用户的反馈产生的场景变化，而动画（animation）则是虚拟世界内部设计的变化。但这两种类型的动态变化之间的界限经常是模糊的。动态的行为既可能源自虚拟世界中图形对象的变化，也可能源自观察者的改变。观察者（照相机或眼睛）自身也可能位于虚拟世界中，并动态地改变自身的位置、方向和其他属性。

Java 3D API提供了完整的基本图形算法的实现，使我们能专注于图形学的主要概念和问题，避免底层实现的烦琐细节。本书将以Java 3D为工具，研究和实现3D图形系统及其应用。

5.3 Java 3D API概述

Java 3D为3D图形场景的建模和绘制提供了高层API。在程序员定义了所有的图形结构和它们的属性之后，Java 3D绘制引擎会自动地绘制整个场景。因此，在Java 3D API之上的编程，仅仅要求程序员定义所需的图形场景和与其相关联的各种属性，而无需实现极其复杂和琐碎的底层绘制过程。

Java 3D利用了Java编程语言的面向对象的特点。图形绘制中所有的元素（例如几何特征、几何变换、光照与动画）都以Java类的形式实现。只需简单地创建这些Java类的对象实例，就能完成这些元素的构造。

Java 3D用一种称为场景图（scene graph）的特殊结构来组织场景绘制中的所有相关对象，它包含了超结构对象、节点对象和节点组件，所有这些一起定义了所要绘制的整个虚拟图形世界。Java 3D绘制引擎将遍历场景图，以持续地进行实际的绘制。场景图定义了一个3D场景的几何属性、外观属性、几何变换、光照属性和视角。同时，场景图还可能包括动画、交互和声音属性。

5.3.1 一个简单示例

程序清单5-1是一个Java 3D的Java小程序（applet）和Java应用程序的例子。这个例子示范了Java 3D程序的基本结构。此程序将显示在一个点光源照射下的立体3D文字“Hello 3D”（见图5-2）。

程序清单5-1 Hello3D.Java

```
1 package chapter5;
2
3 import java.awt.*;
4 import java.applet.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import javax.vecmath.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义Hello3D类，继承自Applet类，演示了点光源下的3D字符串对象
12 public class Hello3D extends Applet {
```

```

13 public static void main(String s[]) {
14     new MainFrame(new Hello3D(), 640, 480); //创建程序主窗口并设置大小
15 }
16 //重写Applet初始化方法
17 public void init() {
18     GraphicsConfiguration gc =
19     SimpleUniverse.getPreferredConfiguration();
20     Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
21     setLayout(new BorderLayout()); //设置布局管理器
22     add(cv, BorderLayout.CENTER);
23     BranchGroup bg = createSceneGraph(); //创建场景图分支
24     bg.compile();
25     SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
26     su.getViewingPlatform().setNominalViewingTransform();
27     su.addBranchGraph(bg);
28 }
29 //生成BranchGroup的私有方法，构造场景图
30 private BranchGroup createSceneGraph() {
31     BranchGroup root = new BranchGroup();
32     //构造图形对象
33     Appearance ap = new Appearance(); //外观对象
34     ap.setMaterial(new Material()); //材质属性
35     Font3D font = new Font3D(new Font("SansSerif", Font.PLAIN, 1),
36                             new FontExtrusion()); //Font3D节点组件
37     Text3D text = new Text3D(font, "Hello 3D"); //Text3D节点组件
38     Shape3D shape = new Shape3D(text, ap); //构造文本Shape3D叶节点
39     //几何变换
40     Transform3D tr = new Transform3D();
41     tr.setScale(0.5); //缩放变换
42     tr.setTranslation(new Vector3f(-0.95f, -0.2f, 0f)); //平移变换
43     TransformGroup tg = new TransformGroup(tr); //几何变换组节点
44     root.addChild(tg);
45     tg.addChild(shape);
46     //设置光照
47     PointLight light = new PointLight(new Color3f(Color.white),
48                                       new Point3f(1f, 1f, 1f),
49                                       new Point3f(1f, 0.1f, 0f)); //添加白色点光源
50     BoundingSphere bounds = new BoundingSphere(); //球体作用范围边界对象
51     light.setInfluencingBounds(bounds); //设置作用范围边界
52     root.addChild(light);
53     return root;
54 }
55 }

```

138

139

这个例子是一个完整的Java 3D程序，该程序显示了内含一个3D字符串“Hello 3D”的一个窗口。你可以看到这些文字的深度效果，以及一个光源投射到这些文字上的光照效果。整个场景的背景是黑色的。

除了AWT包，这个程序还从Java 3D API中导入了以下包：

```

javax.media.j3d
javax.vecmath
com.sun.j3d.utils.universe
com.sun.j3d.utils.geometry

```

javax.media.j3d是Java 3D的主要包。javax.vecmath包则含有对Java 3D很有用的对象类，这

是一些关于向量、矩阵和其他数学对象的类。尽管com.sun.j3d.*包并非Java 3D的核心包，但是这两个包却包含许多可以用来方便地构造基元、视图和Java 3D中其他对象的功能类。

Hello3D程序中所用的可视组件是老式的AWT组件，而非新的Swing组件。例如，我们使用了一个Applet对象而不是JApplet对象，这是因为Java 3D绘制所用的GUI组件Canvas3D是一种重量级组件。尽管重量级组件可以放置在Swing组件JFrame中，但这样做还是有可能在显示过程中导致异常情况，例如可能无法正常显示窗口菜单。

从Applet类继承的Java小程序，既可以实现为一个应用程序，也可以以类似Swing的方式在main方法中，把这个小程序加入到一个Frame类对象中，因为Frame类没有setDefaultCloseOperation方法，窗口的关闭操作需要通过处理WindowEvent事件来实现。类com.sun.j3d.uitls.applet.MainFrame实现了可以把Java小程序当做应用程序运行的功能（第14行）。Hello3D和多数后续的示例程序，都将使用这个方便的功能类来创建双重功能的程序（既能作为应用程序运行，也能作为Java小程序运行）。

在程序的第20行，我们创建了一个Canvas3D类对象，以便显示3D绘制的结果。Canvas3D是AWT组件Canvas类的一个子类，因此我们可以把Canvas3D类对象嵌入到Frame对象内。在createSceneGraph方法中，我们定义了这个程序的场景图。此方法返回一个BranchGroup类对象，我们可以把它关联到一个SimpleUniverse类对象。功能类SimpleUniverse提供了Java 3D绘制的基本框架。把场景图关联到这个SimpleUniverse对象后，整个场景的绘制就开始了。

在createSceneGraph方法中（第30行），我们首先创建了一个BranchGroup类对象作为根节点。需要显示的可视对象是一个3D字符串，这个字符串由一个Text3D对象来表示。此对象的构造需要一个3D字体对象和一个外观对象。我们用Transform3D类来定义几何变换，此对象负责进行缩放和平移变换，这个几何变换将作用在Text3D对象上。我们还创建了一个光源对象，并把它放置在场景中。方法中还定义了一个BoundingSphere类对象，用来把光源的影响范围限制在一个指定区域内。

要完全理解这个Java 3D程序，首先必须理解Java 3D场景图。我们将在下一节详细介绍Java 3D场景图，并在本章的后面对这个示例程序的场景图作完整的分析。

5.3.2 安装Java 3D

要编译和运行一个Java 3D程序，需要一个Java程序开发环境和Java 3D包。Java 3D是Java 2平台的一个可选包，而标准的Java安装是不包含Java 3D的。需要从以下网址下载Java 3D包：

<http://java.sun.com/products/java-media/3D/>

Java 3D在Solaris、Windows、Linux和Mac OS平台上都有相应的发行版本。在Java 2开发平台的基础上安装了Java 3D包之后，就可以编译和运行像程序清单5-1这样的程序了。

Java 3D发行版本包含与平台相关的本地代码和Java类库。其中的Java类封装在四个jar文件中：

```
j3dcore.jar  
j3dutils.jar
```



图5-2 一个显示3D文字的简单Java 3D程序

140


```
j3daudio.jar  
vecmath.jar
```

在Windows平台的安装过程中，这些jar文件通常放置到如下目录之中：

```
<j2sdk directory>/jre/lib/ext
```

与平台相关的本地代码，则实现为三个动态链接库：j3d.dll、j3daudio.dll和j3dutils.dll，它们通常放置在下列目录中：

```
<j2sdk directory>/jre/bin
```

你也可以在诸如JBuilder和NetBeans之类的集成开发环境中编写Java 3D程序，但这样就需要进行额外的配置，然后才能使用Java 3D包的内容。在JBuilder中，需要从Tool菜单的Configure Libraries菜单项创建一个包含以上四个jar文件的库。创建好这个库之后，可以设置工程属性使用这个库。在NetBeans（以及Sun ONE Studio、Forte）中，只要在NetBeans所关联的J2SDK的基础上安装了Java 3D包，就已经可以编译和运行Java 3D程序了。但是，如果想要利用这个集成开发环境的代码完成上下文感知功能，则还需要为Java 3D的jar文件更新Netbeans的解析数据库。只需要添加这四个jar文件，并为其中每个jar文件选择“Update Parser Database”。在一些更新版本的NetBeans上，则不需要这一步。

Java 3D程序可以在安装了Java 3D包的Java运行时环境（Java Runtime Environment, JRE）上运行。Java 3D程序也可以是一个Java小程序（Applet）。如果要在浏览器中运行Java 3D小程序，则需要为浏览器安装Java运行时环境插件，并在此Java运行时环境上安装Java 3D包。

Java 3D通常是在其他的底层图形API，例如OpenGL的基础上开发的。例如，Windows环境下有两个版本的Java 3D：一个版本基于OpenGL，另一个版本基于DirectX。绘制3D图形通常是一个大计算量的任务，为了达到更好的性能，Java 3D试图利用图形硬件提供的图形加速特性。由于不同的显卡、驱动程序及厂商之间的差别很大，在某些平台上使用Java 3D时，可能会遇到一些兼容性问题，以下建议可能对于解决这些问题有所帮助：

- 把视频驱动程序更新到最新的稳定版本。
- 使用最新版本的Java 3D。
- 把图形卡的OpenGL选项的深度缓存，设置为24位或更高。
- 关闭图形卡的硬件加速。
- 加入Java 3D兴趣组：java3d-interest@java.sun.com。

5.4 Java 3D场景图

为了把3D图形绘制中的各种元素有效地组织起来，Java 3D用场景图的概念来构造一个包含所有与3D绘制相关的元素的虚拟世界。场景图是用于场景组织的一个抽象数学模型，它并不是一张图片或是场景的一幅图像。场景图可以在概念上画作一个图表，但它实际的表示，则是在程序中通过对象的创建和方法的调用来完成的。场景图使程序员能够以统一的方式定义复杂的图形结构和动作，也使得Java 3D绘制引擎能够系统而高效地处理3D场景。

141

场景图属于一种被称为有向无环图（directed acyclic graph, DAG）的类似于树的数据结构。每个有向图由一个顶点（或节点）的集合和一些连接这些顶点的有向边（或连接）组成。图5-3示例了一个有6个顶点和8条边的有向图。有向图中的（有向）路径是沿着图中的边移动的一个“顶点-边”序列。例如，图5-3中，b-c-f-e是一条路径。有向图中的环是指图中的封闭路径——即起点与终点相同的路径。例如，图5-3中，a-c-f-e-a是一个环。DAG是不包含任何环的有向图，因此图5-3不是一个DAG。但是，把边e-a去掉之后，图5-3就变成了一个DAG。

(有向) 树 (tree) 是一种特殊的DAG。树从一个节点开始构造，这个节点称为树的根 (root)。从树的根出发，可以有多条边指向其他的节点，这些节点称为根的子节点 (children)。以同样的方式，每个子节点又可以有多条边指向它的子节点。这一过程可以重复任意次数，最终完成一个树的构造。图5-4示例了一棵树。节点a是这棵树的根。在一棵树中，任意一个节点可以有任意个数的子节点 (包括0个)，但是它的父节点只有一个。没有子节点的节点称为叶节点 (leaf)，非叶节点称为内部节点 (internal node)。图5-4中，e、h、c、g是叶节点，a、b、d、f是内部节点。

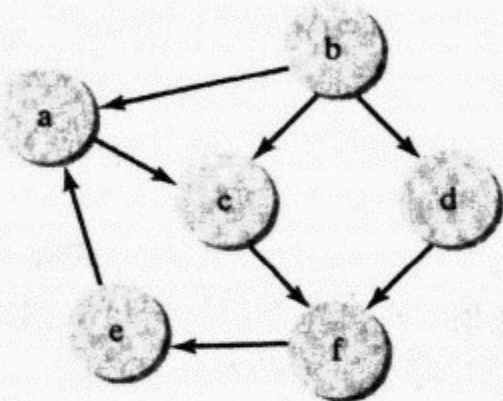


图5-3 一个有向图

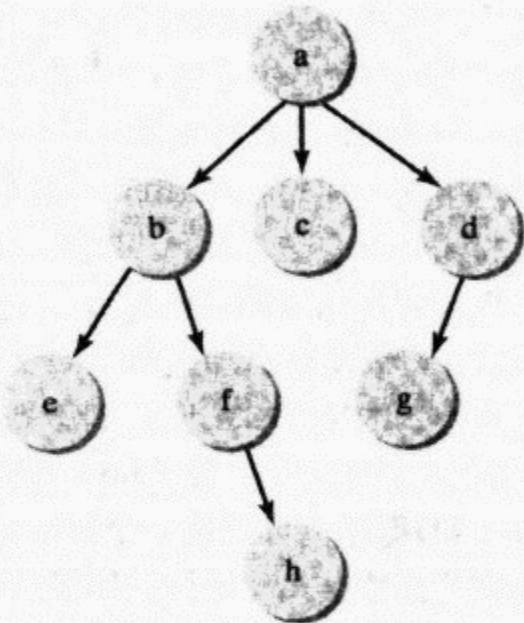


图5-4 一个有向树

场景图的节点代表与图形功能相关的各种类的对象，节点之间的边表示它们之间的逻辑关系。在一个实际的Java 3D程序中，我们通过实例化Java 3D API定义的类或从该API中的类和接口继承的类来创建场景图的节点。节点间的边则通过调用恰当的方法或者类的构造函数来创建。
[142] 图5-5示例了一个非常简单的Java 3D场景图。

在用来图示场景图的图表中，我们用不同的符号来表示不同类型的节点和边。图5-6给出了各种习惯上用来表示场景图中的节点和边的符号。

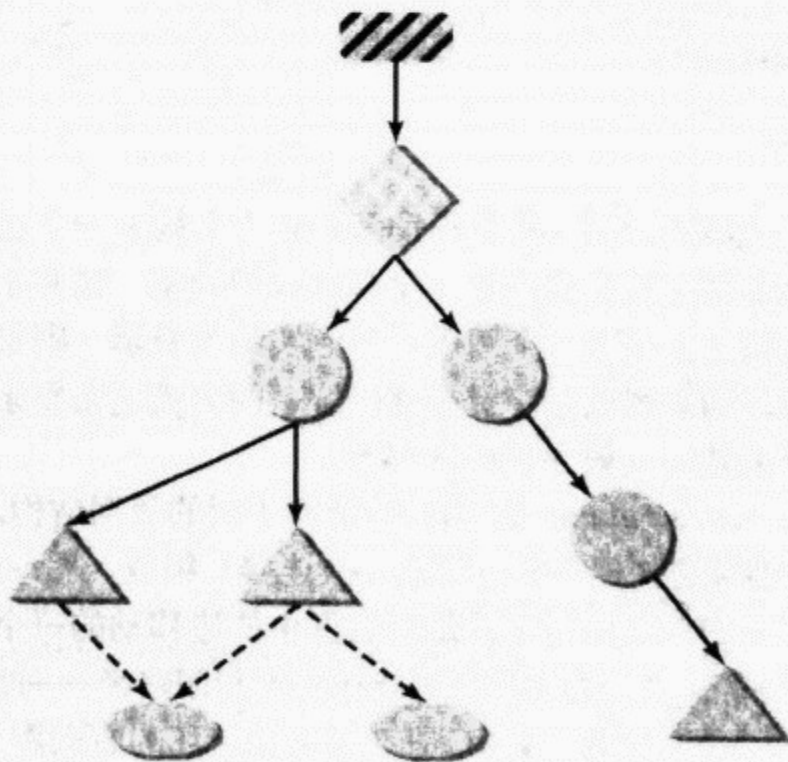


图5-5 一个场景图，它是一个DAG

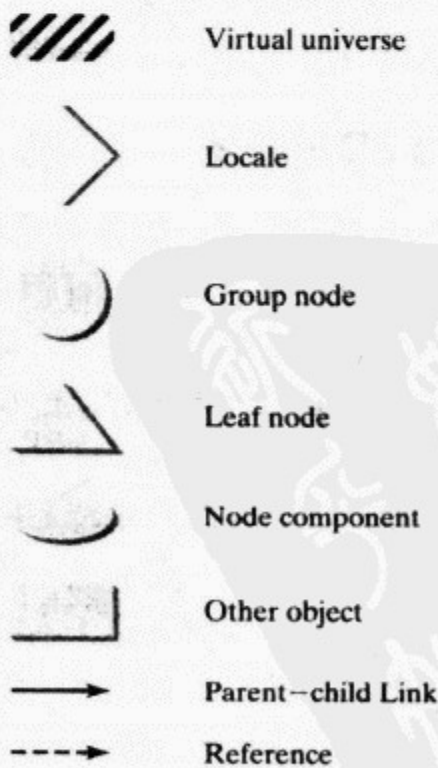


图5-6 场景图的图例

每个场景图主要由三个主要部分构成。在它的顶部是所谓超结构，由VirtualUniverse和Locale类的对象构成。场景图的主体部分是一棵由Node类对象构成的树。第三部分则是一些NodeComponent（节点组件）类对象。树结构中的叶节点可以引用节点组件对象，一个NodeComponent类对象可以被多个叶节点对象引用，因此，整个场景图的结构并非是一棵树，而只是一个DAG。

图5-7给出了场景图中的元素的主要的类层次。VirtualUniverse和Locale类是用于超结构的类，它们不是从SceneGraphObject抽象类继承的。场景图中的树节点都由抽象类Node的子类定义，NodeComponent抽象类则是各种节点组件类的基类。

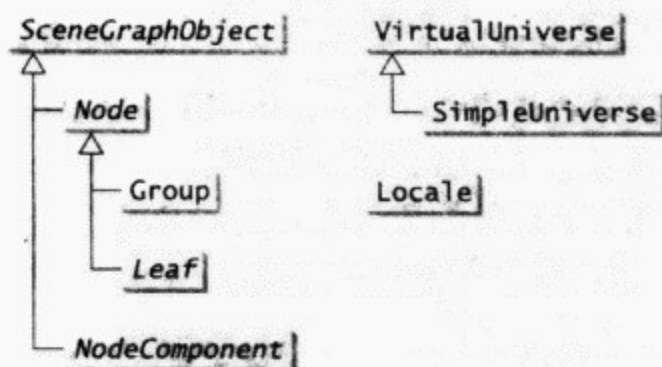


图5-7 场景图的类层次

143

5.5 超结构

VirtualUniverse类和Locale类对象是场景图的超结构对象。每个Java 3D程序通常只有一个VirtualUniverse对象，我们用VirtualUniverse对象来表示任何Java 3D程序都要用到的整个3D空间。为了能容纳整个“宇宙”（universe）的大小和精度，VirtualUniverse类对象用三个高精度的256位定点数来表示它的坐标。这样高精度的256位定点数的二进制小数点在256位的中间，因此它的整数部分有128位，小数部分也有128位。如果以1.0表示一米的单位长度，这样的定点数可以描述长达 2^{127} 米的距离，并且其描述精度高达 2^{-128} 米，这已经足以描述宇宙间的任何真实物体了。例如，地球到太阳之间的距离也仅有 2^{37} 米，而据估计一个质子的直径约为 2^{-50} 米。

我们用类HiResCoord来表示这样的高精度坐标。这样一个HiResCoord对象包含三个256位高精度定点数，分别表示空间中某个位置的x、y、z坐标。

虽然VirtualUniverse类能够用HiResCoord数模拟我们所知的整个宇宙，但用HiResCoord类对象来表示所有点的坐标显然效率太低。因此，Java 3D用Locale类对象来表示一个较小的局部空间，这样一来就大大提高了表示效率。每个Locale类对象定义一个局部坐标系，这个局部坐标系被锚定在虚拟宇宙中一个由HiResCoord对象指定的位置上。在一个局部坐标系内部，每个点的坐标则用普通的浮点数表示。每个VirtualUniverse可以包含一个或多个Locale对象，每个Locale对象都可以附属有分支图。当一个分支图被附属到一个Locale时，Java 3D绘制引擎就开始绘制此分支图，此分支图则成为活动分支图。每个Locale对象总是附属属于一个VirtualUniverse对象，这种关联是通过Locale对象的构造函数建立的。

```

Locale(VirtualUniverse vu)
Locale(VirtualUniverse vu, HiResCoord location)
Locale(VirtualUniverse vu, int[] x, int[] y, int[] z )
  
```

每个Locale对象在虚拟宇宙中的位置，可以用一个HiResCoord对象定义，也可以由三个整形数组来指定，默认的位置是 (0, 0, 0)。以下语句将创建场景图的超结构：

```

VirtualUniverse universe = new VirtualUniverse();
Locale locale = new Locale(universe);
  
```

可以用以下方法把一个以BranchObject类对象为根的场景图分支附属到一个Locale对象上：

```
void addBranchGraph(BranchGroup branch)
```

可以用以下方法对场景图的分支进行修改：

```

void replaceBranchGraph(BranchGroup oldBranch, BranchGroup newBranch)
void removeBranchGraph(BranchGroup branch)
  
```

144

用以下方法可以获得任何一个Locale对象的分支的数目及其所有分支对象:

```
int numBranchGraphs()
Enumeration getAllBranchGraphs()
```

SimpleUniverse类是从VirtualUniverse类继承的一个功能类, 它包含一个Locale对象和一组定义了一个标准视图的对象。把SimpleUniverse对象和可视内容分支进行组合, 就可以快速地构成一个完整的场景图。Java 3D中的世界坐标系统是一个右手直角坐标系, 默认的观察位置在z轴上, 观察方向为z轴的负方向。从观察者的视角看, x轴指向右侧而y指向上方。

5.6 节点

Node类对象是场景图中的主体树结构的节点, 主要有两种节点对象: Group节点和Leaf节点。Group节点是树的内部节点, 通常表示其子节点之间的某种特定关系, 或者表示某种作用于其子节点的操作。Leaf节点是树的叶节点, 表示具体的图形实体。场景图中的叶节点通常会通过引用一些NodeComponent对象来定义其特定属性, 多个叶节点之间可能会共享某些NodeComponent对象。

5.6.1 组节点

组(group)节点是场景图中的内部节点, 图5-8给出了组节点类的主要类层次。组节点是场景图之树结构的主要构建模块(building blocks)。组节点可能会有子节点, 组节点的子节点可以是叶节点, 也可以是又一个组节点。因为场景图中节点对象必须形成树结构, 所以两个组节点之间不能共享同一子节点对象。每个子节点仅有一个父节点, 且从根节点到每一个叶节点都只有一条唯一的路径。

每个Group节点都保存了一个属于自己的子节点列表。要给一个组节点添加一个子节点, 可以调用以下方法:

```
void addChild(Node child)
void insertChild(Node child, int index)
```

可以通过索引号访问某个子节点:

```
Node getChild(int index)
void setChild(Node child, int index)
```

可以用以下方法查询一个组节点的子节点的情况:

```
int numChildren()
int indexOfChild(Node child)
```

其他可以和一个组节点的子节点相关的方法还包括:

```
Enumeration getAllChildren()
void removeChild(Node child)
void removeChild(int index)
void removeAllChildren()
```

每个BranchGroup节点都是场景图的某一个分支的根节点, 只有BranchGroup节点才能直接附属到一个Locale类对象上, 因此, 每个场景图中至少有一个BranchGroup节点。每个BranchGroup节点仅仅完成把它的子节点集合到一起的功能, 并没有其他特殊的作用。

OrderedGroup类节点可以指定其子节点的绘制顺序。Java 3D绘制引擎从根节点开始遍历场景图, 直到每个叶节点, 在此过程中绘制整个场景。通常在任意一个节点上, 其子节点的访

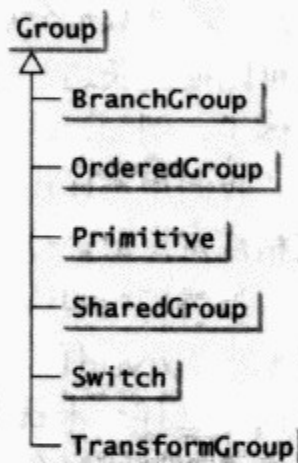


图5-8 组节点类

问顺序是不确定的。Java 3D引擎可能以任何一种顺序来逐个绘制这个节点的每一个子节点，用OrderedGroup类节点则可以保证以确定的顺序来遍历其子节点。OrderedGroup类节点保证其子节点按照它们的索引号顺序被逐个绘制，例如在以下代码中，我们构造了一个场景图的一部分，其中的三个形状将保证按照shape1、shape2、shape3的顺序逐个进行绘制，相应的场景图如图5-9所示。

```
Shape3D shape1 = new Shape3D();
Shape3D shape2 = new Shape3D();
Shape3D shape3 = new Shape3D();
OrderedGroup group = new OrderedGroup();
group.addChild(shape1);
group.addChild(shape2);
group.addChild(shape3);
```

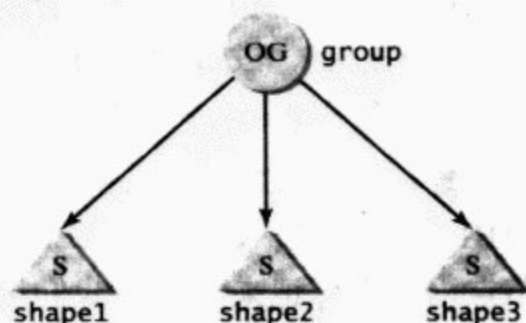


图5-9 一个OrderedGroup节点及其子节点

可以用Primitive类节点表示诸如球体之类的完整的几何基元。Primitive类是com.sun.j3d.utils.geometry包中定义的一个功能类，我们将在第6章中讨论几何基元的相关内容。

SharedGroup类节点作为一个分支图的根节点时，可以被多个Link类的叶节点所共享。我们经常会遇到这样的情况，即在一个场景图中，有多个分支是完全相同的，但是普通的分支是不能被组节点所共享的，因为我们要求每个分支必须是一个树结构。在这种情况下，可以把这些相同的分支定义为同一个分支，并以一个SharedGroup类的对象作为其根节点。然后，多个Link类的叶节点就能通过引用关系而不是父子节点关系来共享这个分支。考虑图5-10中的简单例子，左边这个场景图中，两个分支有完全相同的结构和属性。为了共享这一分支，我们加入了一个SharedGroup类节点作为共享分支的根节点，并引入了两个Link类节点代替了原来的两个分支，这两个Link类节点都引用这个共享分支，于是就得到了右边图中所示的场景图。注意，这个结构并没有违反树结构的要求，因为SharedGroup节点并非Link节点的子节点。

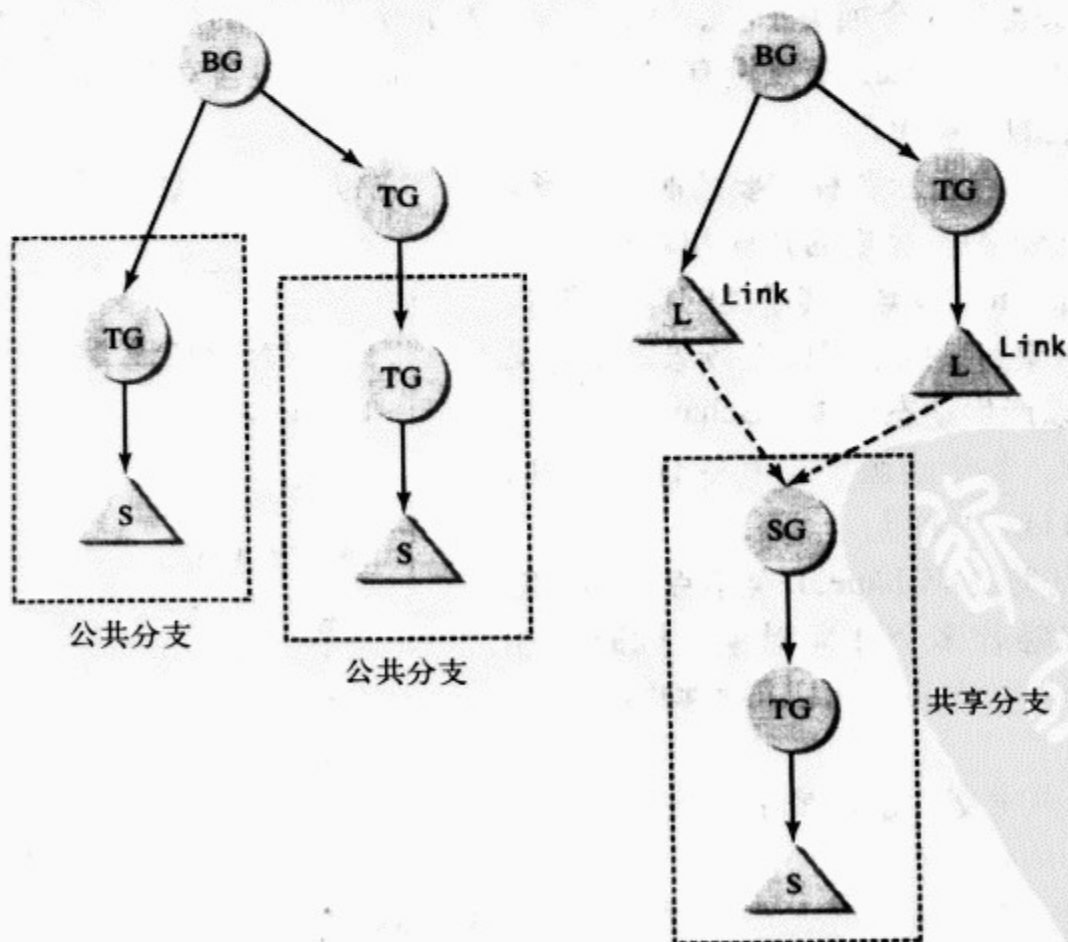


图5-10 用SharedGroup节点和Link叶节点来共享相同的分支

Switch类节点可以作为一个选择器，来选择一组特定的子节点进行绘制。在一个Switch节点的所有子节点中，可以选择绘制其中的某一个子节点，或者绘制所有的子节点，或者一个子节点也不绘制，或者用下面的方法选择绘制其中一组子节点：

```
void setWhichChild(int whichChild)
```

参数whichChild可以是一个表示子节点索引的非负整数，或者是以下预定义常数中的一个：CHILD_NONE、CHILD_ALL、CHILD_MASK。如果使用常数CHILD_MASK，则被选中的子节点由一个标记来定义，我们可以用下面的方法来设置这个标记：

147

```
void setChildMask(BitSet mask)
```

例如，以下这段代码选择对shape1和shape3进行绘制：

```
Shape3D shape1 = new Shape3D();
Shape3D shape2 = new Shape3D();
Shape3D shape3 = new Shape3D();
Switch group = new Switch();
group.addChild(shape1);
group.addChild(shape2);
group.addChild(shape3);
BitSet mask = new BitSet();
mask.set(0);
mask.set(2);
group.setChildMask(mask);
group.setWhichChild(Switch.CHILD_MASK);
```

TransformGroup类节点表示一个将作用在其所有子节点上的几何变换，TransformGroup类节点用一个Transform3D对象来定义其几何变换。我们将在第7章中介绍几何变换的相关内容。

5.6.2 叶节点

Leaf类是Node类的一个抽象子类。通常Leaf（叶）节点表示的是各种几何对象、声音或场景图中的其他图形对象。Leaf节点没有子节点，但它们通常都包含对节点组件对象的引用。图5-11给出了叶节点类层次。

Shape3D类叶节点表示各种需要绘制的图形对象。它包含对节点组件的引用，而这些节点组件则定义了Shape3D类对象的几何和外观属性。可以在第6章找到Shape3D类及其子类的详细信息。

Behavior类对象封装的是可以在场景图中定义的动作，通常用以产生动态效果。Behavior类是动画和交互的基础。关于动画和交互的详细信息，将在第10章和第11章中介绍。

Morph类节点类似于Shape3D类节点，不同的是，可以用它来融合多个几何对象，同时使用Morph节点和Behavior节点，可以得到物体变形的效果。

Light类叶节点定义的是场景中的光源，我们将在第9章中讨论光照模型。

Fog类节点能提供一种特殊绘制效果，即把对象的颜色和另一种颜色进行混合，混合的程度与对象和观察者之间的距离有关，通过这种混合可以营造出淡出或雾化的效果。我们将在第9章中介绍Fog类的细节。

148

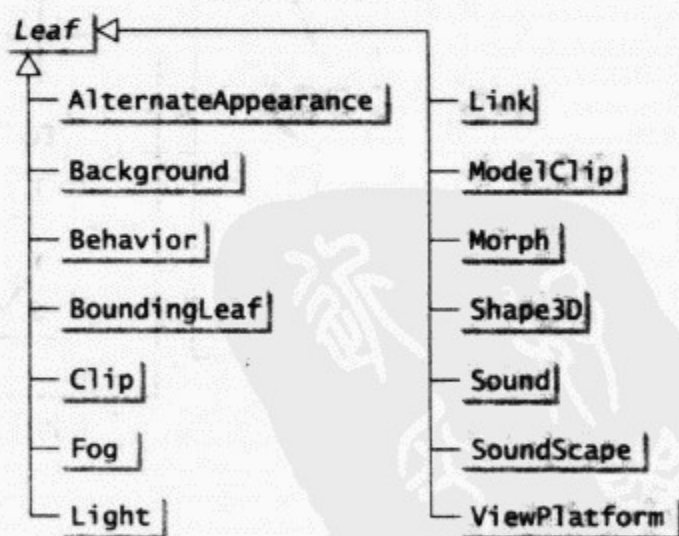


图5-11 叶节点类

我们用ViewPlatform类节点来表示观察者在场景图中的位置。ViewPlatform类是Java 3D复杂的观察系统的一部分，并和一个View类对象相关联。我们将在第8章中讨论视图的相关内容。

我们可以用Background类对象来定义一个场景的背景。另外，在本章后半部分将讨论如何使用Background类。

我们用Clip类和ModelClip类对象来定义用以裁剪视图的平面。Clip节点定义的是一个远端裁剪平面，在此平面背后的任何对象都将被裁剪掉，并从绘制过程中排除，而通过使用ModelClip节点，可以定义六个平面来完成一个视图的裁剪。

如前面已经介绍过的，可以用Link类叶节点来引用SharedGroup类对象，以达到共享场景图中相同和重复的分支的目的。

通过使用AlternateAppearance类节点，可以覆盖一个可视对象的外观。如果一个Shape3D对象或Morph对象处在一个AlternateAppearance类节点的影响范围内，而它们又允许自己的外观被覆盖的话，AlternateAppearance对象就可以覆盖它们的外观。

Sound和SoundScape类节点表示的是声音对象。Java 3D允许在场景图中嵌入声音，对于游戏视频之类的应用来说，这是一个很有用的功能。

5.7 节点组件

我们用组节点和叶节点来定义一个场景的结构，但这些节点的属性则通常由其他对象另外定义，绝大多数用来定义属性的对象都属于NodeComponent类（节点组件类）。可以用NodeComponent类对象来定义几何特征、颜色、纹理和材质等属性。NodeComponent类对象本身并非场景图的主体树结构的节点，通常它们只是被场景图中的叶节点所引用。图5-12给出了NodeComponent类及其子类的类层次。

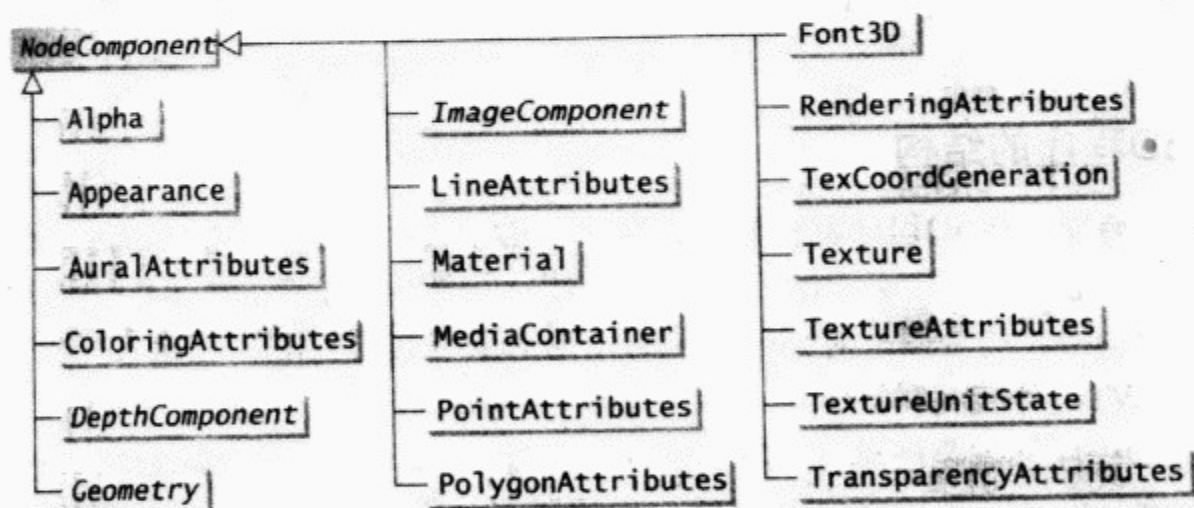


图5-12 NodeComponent类

Geometry类（及其子类）对象用来定义图形对象的几何属性，每个Shape3D叶节点都依赖于一个Geometry对象来定义其几何特征。

Appearance类对象用来控制所绘制对象的外观，而Appearance类对象则含有对其他属性对象的引用。Appearance类对象定义了一个Shape3D叶节点的完整的绘制状态，它所引用的其他属性对象包括ColoringAttributes、TransparencyAttributes、Material、PointAttributes、LineAttributes、PolygonAttributes、RenderingAttributes、Texture、TextureAttributes和TexCoordGeneration等。我们将在第6章讨论Geometry类和Appearance类。

ColoringAttributes类对象用来定义绘制一个可视对象时所用的颜色和光照模型。

同时，TransparencyAttributes类对象用来设置图形对象的透明属性，包括它的透明模式、混合函数及混合值。

此外, Material类对象用来定义那些比ColoringAttributes类对象所定义的更为复杂的材质属性, Material类对象通常在光照模型中使用。我们将在第9章详细讨论材质属性和使用材质属性可以达到的效果。

RenderingAttributes类对象用来定义一些与绘制相关的参数, 例如深度缓存和alpha测试等。

PointAttributes类、LineAttributes类和PolygonAttributes类对象用来定义与点、线和多边形对象的绘制相关的属性。点属性包括点的大小和反走样设置, 线属性包括线宽、线型和反走样设置, 多边形属性则包括与多边形的绘制相关的属性, 例如多边形绘制模型、裁剪(culling)、背部法线反转(backface normal flip)和偏移(offset)等。

Texture、TextureAttributes和TextureUnitState是一组与纹理映射绘制技术相关的类。应用纹理映射技术, 可以用图像来绘制可视对象的细节, 用于映射的图像封装于ImageComponent类对象内部, ImageComponent类对象还可以用于设置场景背景。纹理坐标生成是纹理映射技术的重要组成部分, 而TexCoordGeneration类对象则可以用于纹理坐标的自动生成。我们将在第9章和第12章介绍纹理映射。

AuralAttributes类对象定义了一些与声效呈现相关的参数。我们用MediaContainer类对象来定义声音数据, 我们将在第12章给出在Java 3D中使用声音效果的例子。

Alpha类对象用于把时间值转换为0到1之间的浮点数(alpha值)。Alpha类的作用是作为信号或波形发生器, 以激发某些行为, 这在动画中非常有用。我们将在第11章中介绍动画的相关内容。

DepthComponent类对象用于封装深度缓存的概念(也称为z缓存)。深度缓存是一个用于存储深度值(z值)的2D数组。

Fond3D类对象用来定义实体3D字体。每个Font3D对象都是以一个AWT Font类对象和FontExtrusion类对象为基础的, AWT Font类对象用于定义字体的2D字型, 而FontExtrusion类对象则在第3D上定义拉伸路径。

5.8 Java 3D程序的结构

简单说来, 编写Java 3D程序就相当于装配一个场景图。当然, 我们也需要创建通常的用户界面元素和其他一般应用程序必需的元素。但是对一个图形程序来说, 3D建模和绘制的主体部分还是场景图的构造。场景图是对系统中所有图形对象及其属性的完整定义, 并和用于显示场景绘制图像的AWT组件相关联。

每个Java 3D程序都需要创建一个Canvas3D类对象。Canvas3D是java.awt.Canvas类的一个子类, 所以其实例对象可以和其他AWT组件一样放置在AWT容器内。Canvas3D对象就像是一个用于显示虚拟世界场景的绘制结果的画布。

Java 3D程序必须用Java 3D API所提供的类或继承的类来构造出完整且正确的场景图。一个VirtualUniverse类对象和一个Locale对象是场景图所必需的超结构。

需要一个观察分支来建立场景的视图。这个分支通常由BranchGroup类、TransformGroup类、ViewPlatform类、View类、PhysicalBody类和PhysicalEnvironment类对象组成。观察分支附属于一个Locale类对象, 并关联到一个Canvas3D类对象, 以显示视图的绘制结果。

除了观察分支之外, 场景图至少还需要另外一个分支来表示虚拟世界的图形内容。这个内容分支应该以一个BranchGroup节点为根, 这样才能附属到场景图的Locale节点上, 其他的节点则可以附属到此BranchGroup节点上来完成虚拟世界的构造。我们可以用Shape3D类、Light类和其他类节点来创建图形对象, 每个Shape3D节点的创建, 都必须引用恰当的Geometry类和Appearance类对象来指定其几何及外观属性, TransformGroup节点则用来对其子节点施加必要

的几何变换。

为了简化场景图的构造过程，Java 3D提供了一个方便的功能类SimpleUniverse。SimpleUniverse类会创建一个VirtualUniverse对象、一个Locale对象以及一个标准视图所需的对象。对于多数使用普通视频显示设备的Java 3D应用来说，使用SimpleUniverse类对象就可以完成超结构和视图的构造和设置。

对Java 3D场景图有了基本的了解后，我们就可以对程序清单5-1中的程序进行仔细的分析。图5-13所示是与这个程序对应的场景图。

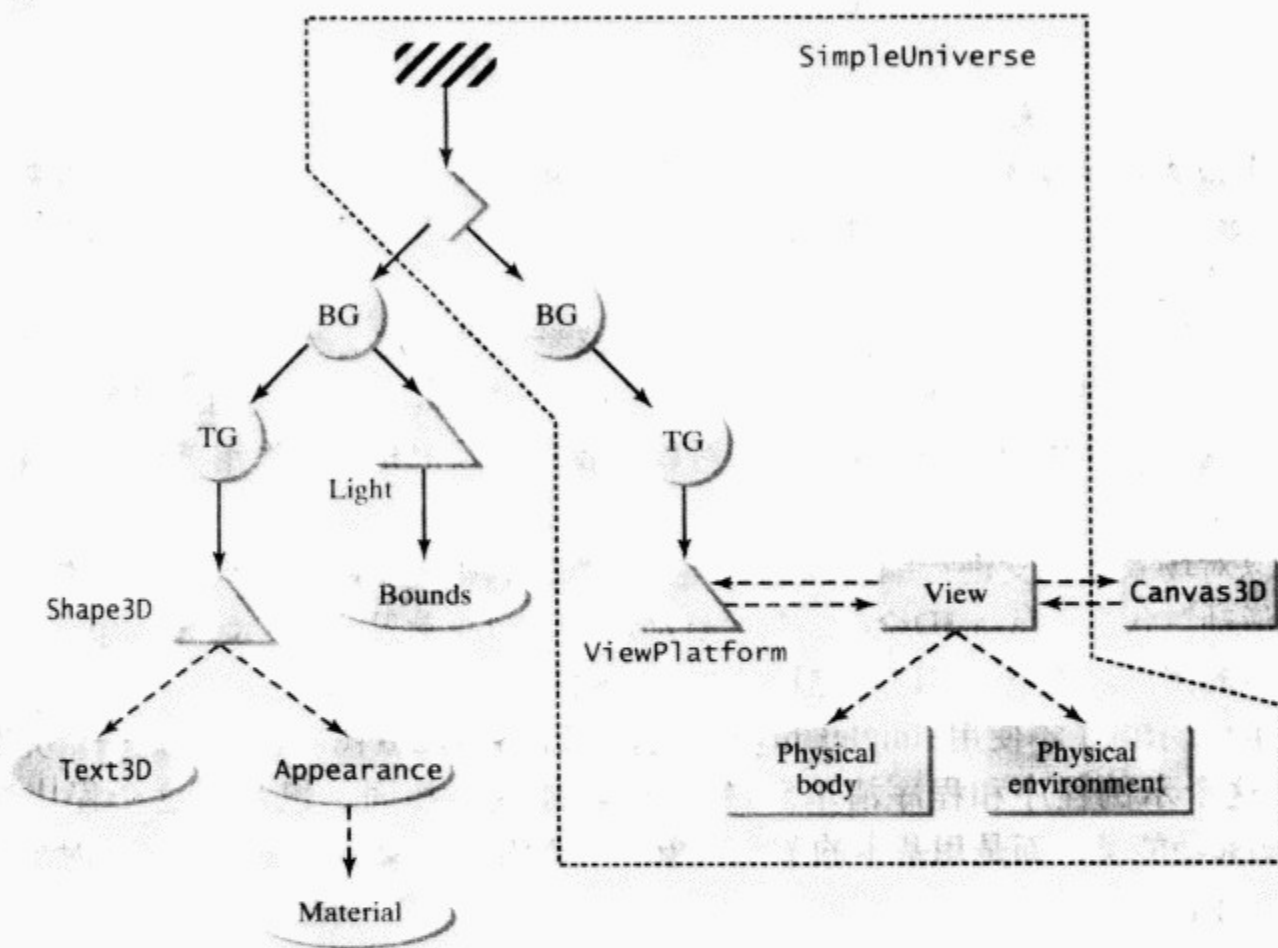


图5-13 程序清单5-1的场景图

SimpleUniverse类对象内部包含了VirtualUniverse对象、Locale对象、BranchGroup对象、TransformGroup对象、ViewPlatform对象、View对象、PhysicalBody对象和PhysicalEnvironment对象。场景图中除了SimpleUniverse对象之外的分支，就是定义虚拟世界中图形对象的内容分支。通过调用SimpleUniverse对象的attachBranchGraph()方法，我们可以把内容分支附属到Locale对象上。

SimpleUniverse类对象内的View对象与一个Canvas3D对象相关联。Canvas3D对象是一个AWT组件，Canvas3D对象需要另外创建，并添加到一个AWT容器中，以显示图形系统的绘制结果。SimpleUniverse对象中的View对象定义了一组默认参数。下面列出了部分参数：

投影方式	PERSPECTIVE_PROJECTION（透视投影）
视角范围	$\pi/4$
前端裁剪距离	0.1
后端裁剪距离	10.0

PhysicalBody对象和PhysicalEnvironment对象，也是以适用于普通计算机屏幕视图的默认参数创建的。SimpleUniverse对象内的场景图的观察分支由一个BranchGroup节点、一个TransformGroup节点和一个ViewPlatform叶节点组成。因为观察分支要附属到Locale对象上，所以BranchGroup（BG）对象是必需的。TransformGroup（TG）节点为ViewPlatform定义了一个

几何变换，默认情况下的视图的观察平面通过坐标原点。某些情况下，场景中的一些物体碰巧也离坐标原点较近，这样会使观察不便，避免这种问题的办法是改变视图的几何变换，沿z轴向后移动视图。只需调用ViewingPlatform对象的setNominalViewingTransform方法，就可以完成这种改变。程序清单5-1中，以下这一行从SimpleUniverse对象获取ViewingPlatform对象，并调用其方法以移动视图：

```
su.getViewingPlatform().setNominalViewingTransform();
```

这个程序的内容分支以一个BranchGroup节点BG作为根。BG左侧的子节点是一个TransformGroup节点TG。该节点将对其所有子节点施加一个仿射变换，该仿射变换包括一个系数为0.5的缩放变换和一个 $(-0.95, -0.2, 0)$ 的平移变换，TG节点只有一个子节点，这个子节点是一个Shape3D类对象，这个Shape3D对象所引用的几何属性对象是一个Text3D类的对象，将该Text3D节点组件设置为显示文字“Hello 3D”。Shape3D对象所引用的外观属性则是一个默认参数的Material类对象，把外观属性设置为Material对象，将激活光照模式，此3D文字对象将被场景中定义的光源照亮。

BG节点的另一个子节点是一个光源节点，该光源对象定义为一个白色点光源，位置坐标为 $(3, 3, 3)$ ，光强衰减系数为 $(1, 0, 0)$ 。为了提高运算效率，我们可以限制该光源的作用范围。由于这个程序中只有一个Shape3D对象，我们可以把光源的作用范围限制到只需覆盖此3D文字对象。

我们可以对场景图分支进行编译以提高运行时性能。把分支附属到Locale对象之后，此分支就变成“活动”状态，Java 3D绘制引擎会自动地对活动场景图（live scene graph）进行绘制。活动场景图中的图形对象就不可以再进行编辑了，除非经过特别允许。

程序清单5-2示例了不使用SimpleUniverse类，而用各种场景图对象来构造一个完整的场景图的过程。这个示例程序和程序清单5-1完成的功能完全相同。但是，这个程序没有使用SimpleUniverse功能类，而是用基本的节点对象、节点组件对象和其他相关类的对象来构造了一个完整的场景图。

程序清单5-2 Hello3DfullGraph.java

```
1 package chapter5;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.media.j3d.*;
6 import javax.vecmath.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义Hello3DfullGraph类，继承自Applet类，演示完整场景图的构造
12 public class Hello3DFullGraph extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new Hello3DFullGraph(), 640, 480); //创建主窗口并设置大小
15     }
16     //重写Applet初始化方法
17     public void init() {
18         //创建Canvas3D画布对象
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
```



```

21 Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
22 setLayout(new BorderLayout()); //设置布局管理器
23 add(cv, BorderLayout.CENTER);
24
25 //创建超结构
26 VirtualUniverse vu = new VirtualUniverse();
27 Locale loc = new Locale(vu);
28
29 //创建视图分支
30 BranchGroup bgView = createViewBranch(cv); //创建视图分支
31 bgView.compile();
32 loc.addBranchGraph(bgView); //将该视图分支附属到Locale节点
33
34 //创建场景内容分支
35 BranchGroup bg = createContentBranch(); //创建场景图分支
36 bg.compile();
37 loc.addBranchGraph(bg); //将该场景图分支附属到Locale节点
38 }
39 //生成BranchGroup的私有方法, 创建视图分支
40 private BranchGroup createViewBranch(Canvas3D cv) {
41     View view = new View(); //创建View组件对象
42     view.setProjectionPolicy(View.PERSPECTIVE_PROJECTION); //投影方式
43     ViewPlatform vp = new ViewPlatform(); //ViewPlatform叶节点
44     view.addCanvas3D(cv);
45     view.attachViewPlatform(vp);
46     view.setPhysicalBody(new PhysicalBody());
47     view.setPhysicalEnvironment(new PhysicalEnvironment()); //设置view对象属性
48     Transform3D trans = new Transform3D(); //几何变换
49     Point3d eye = new Point3d(0, 0, 1/Math.tan(Math.PI/8)); //观察者眼睛位置
50     Point3d center = new Point3d(0, 0, 0); //观察方向指向的点
51     Vector3d vup = new Vector3d(0, 1, 0); //垂直于观察方向向上的方向
52     trans.lookAt(eye, center, vup); //生成几何变换矩阵
53     trans.invert(); //求矩阵的逆
54     TransformGroup tg = new TransformGroup(trans); //几何变换组节点
55     tg.addChild(vp);
56     BranchGroup bgView = new BranchGroup(); //创建视图分支
57     bgView.addChild(tg);
58     return bgView;
59 }
60 //创建场景内容分支
61 private BranchGroup createContentBranch() {
62     BranchGroup root = new BranchGroup(); //创建场景图分支
63     //构造图形对象
64     Appearance ap = new Appearance(); //外观对象
65     ap.setMaterial(new Material()); //材质属性
66     Font3D font = new Font3D(new Font("SansSerif", Font.PLAIN, 1),
67                             new FontExtrusion()); // Font3D节点组件
68     Text3D text = new Text3D(font, "Hello 3D"); //Text3D节点组件
69     Shape3D shape = new Shape3D(text, ap); // Shape3D叶节点
70     //几何变换
71     Transform3D tr = new Transform3D(); //几何变换
72     tr.setScale(0.5); //缩放变换
73     tr.setTranslation(new Vector3f(-0.95f, -0.2f, 0f)); //平移变换
74     TransformGroup tg = new TransformGroup(tr); //几何变换组节点

```



```

75     root.addChild(tg);
76     tg.addChild(shape);
77     //设置光源
78     PointLight light = new PointLight(new Color3f(Color.white),
79                                     new Point3f(1f,1f,1f),
80                                     new Point3f(1f,0.1f,0f)); //添加白色点光源
81     BoundingSphere bounds = new BoundingSphere();//球体作用范围边界对象
82     light.setInfluencingBounds(bounds); //设置作用范围边界
83     root.addChild(light);
84     return root;
85 }
86 }

```

153

这个例子与程序清单5-1中的例子程序实质上是相同的，它构造的场景图和程序清单5-1构造的场景图完全一样，如图5-13所示。它们的区别在于，这个例子没有使用SimpleUniverse类，而是显式地构造了场景图的超结构和观察分支。

程序的第26到27行以默认参数创建了超结构的VirtualUniverse和Locale对象。Locale对象位于默认坐标 (0, 0, 0)，内容分支与程序清单5-1中完全相同。观察分支的创建与程序清单5-1中不同，它用createViewBranch方法（第40行）创建。如图5-13所示，观察分支由BranchGroup对象、TransformGroup对象、View对象、ViewPlatform对象、PhysicalBody对象和PhysicalEnvironment对象构成。

TransformGroup对象用一个Transform3D对象来表示其几何变换，lookAt方法和invert方法（第52到53行）可以用来构造几何变换。lookAt方法基于观察者眼睛的位置、观察方向和垂直于观察方向的向上方向设置一个几何变换，这个几何变换的逆变换最终用于设置视图的TransformGroup对象。在这个示例程序中设置的几何变换和上一个示例程序中SimpleUniverse对象通过调用setNominalViewingTransform方法设置的几何变换是相同的，因此这个示例程序的显示结果与程序清单5-1的显示完全相同，如图5-2所示。

5.9 背景和边界

场景的默认背景是一个黑色背景，如果在场景图中未放入任何可视对象或光源的话，将看到一片漆黑。可以用Background类叶节点来改变场景背景，可以用Background叶节点定义单色的背景，也可以定义一幅背景图像，甚至还可以用几何的方式来定义背景。背景总是绘制在所有图形对象之后。以下列出的是Background类的构造函数：

```

Background()
Background(Color3f color)
Background(float r, float g, float b)
Background(ImageComponent2D image)
Background(BranchGroup geometry)

```

程序清单5-3中使用了Background类节点。这个程序类似于程序清单5-1，但绘制的背景是白色而不是黑色的。

程序清单5-3 Hello3Dbackground.java

```

1 package chapter5;
2
3 import java.awt.*;
4 import java.applet.*;
5 import java.awt.event.*;

```



```
6 import javax.media.j3d.*;
7 import javax.vecmath.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义Hello3Dbackground类, 继承自Applet类, 演示Background节点的使用
12 public class Hello3DBackground extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new Hello3DBackground(), 640, 480); //创建主窗口并设置大小
15     }
16     //重写初始化方法
17     public void init() {
18         GraphicsConfiguration gc =
19             SimpleUniverse.getPreferredConfiguration();
20         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
21         setLayout(new BorderLayout()); //设置布局管理器
22         add(cv, BorderLayout.CENTER); //画布对象位置居中
23         BranchGroup bg = createSceneGraph(); //创建场景图分支
24         bg.compile(); //编译场景图
25         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
26         su.getViewingPlatform().setNominalViewingTransform();
27         su.addBranchGraph(bg);
28     }
29     //生成BranchGroup的私有方法, 创建场景图分支
30     private BranchGroup createSceneGraph() {
31         BranchGroup root = new BranchGroup();
32         //构造图形对象
33         Appearance ap = new Appearance(); //外观对象
34         ap.setMaterial(new Material()); //材质属性
35         Font3D font = new Font3D(new Font("SansSerif", Font.PLAIN, 1),
36             new FontExtrusion()); //Font3D节点
37         Text3D text = new Text3D(font, "Hello 3D"); //Text3D节点组件
38         Shape3D shape = new Shape3D(text, ap); //Shape3D叶节点
39         //几何变换
40         Transform3D tr = new Transform3D();
41         tr.setScale(0.5); //缩放变换
42         tr.setTranslation(new Vector3f(-0.95f, -0.2f, 0f)); //平移变换
43         TransformGroup tg = new TransformGroup(tr); //几何变换组节点
44         root.addChild(tg);
45         tg.addChild(shape);
46         //设置光源
47         PointLight light = new PointLight(new Color3f(Color.white),
48             new Point3f(1f, 1f, 1f),
49             new Point3f(1f, 0.1f, 0f)); //添加白色点光源
50         BoundingSphere bounds = new BoundingSphere(); //球体作用范围边界对象
51         light.setInfluencingBounds(bounds); //设置作用范围边界
52         root.addChild(light);
53         //设置背景
54         Background background = new Background(1.0f, 1.0f, 1.0f);
55         background.setApplicationBounds(bounds); //设置作用范围边界
56         root.addChild(background);
57         return root;
58     }
59 }
```


这个例子和程序清单5-1是相似的，唯一的区别是其背景变成了白色，如图5-14所示。



图5-14 一个使用白色背景的简单Java 3D程序

图5-15给出了新的场景图，内容分支上新加入了一个Background类叶节点（第54到56行），创建背景对象的构造函数指定了背景的RGB颜色：

```
Background background = new Background(1.0f, 1.0f, 1.0f);
background.setApplicationBounds(bounds);
root.addChild(background);
```

背景对象的作用范围边界是和光源对象共享的一个BoundingSphere类对象。

诸如Background类和Light类的环境类节点可以影响整个虚拟宇宙，但是为了达到合理的绘制效率，我们需要限制这些元素的作用范围。可以用两种不同的方式指定环境节点作用范围的边界（bounds）：使用Bounds类对象或使用BoundingLeaf类对象。Bounds对象属于节点组件，而BoundingLeaf类对象则属于场景图的叶节点，这两种方式的主要区别在于边界的坐标。直接引用Bounds类对象来设置边界的环境类节点所设置的边界位置，是相对于此环境类节点自身的。BoundingLeaf类节点则定义了自身的位置坐标，引用BoundingLeaf类节点的环境类节点所获得的边界位置，是相对于BoundingLeaf节点的坐标的。

155

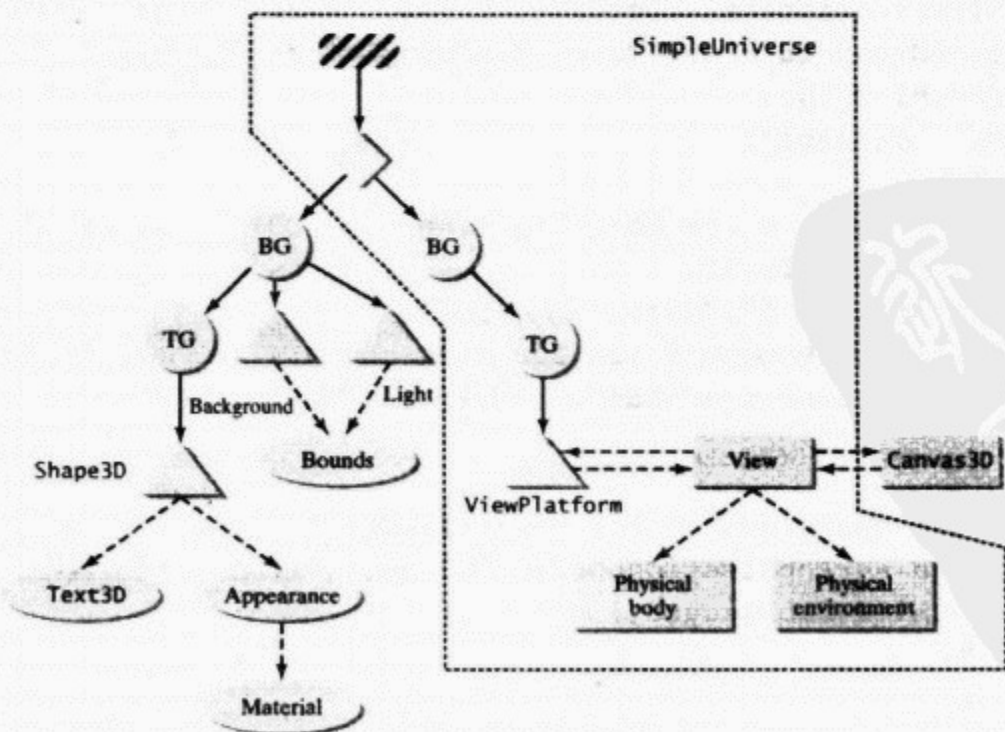


图5-15 程序清单5-3的场景图

图5-16所示为Bounds类的类层次结构。抽象类Bounds有三个子类，分别是BoundingBox类、BoundingSphere类和BoundingPolytope类，可以用来表示不同形状

156

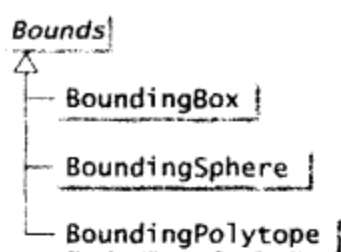


图5-16 Bounds类层次

边界。BoundingBox类定义的是一个立方体盒的边界，立方体的各条边则与坐标轴平行。BoundingSphere类定义的是一个球形边界，BoundingPolytope类定义的则是一个一般性的多面体边界。

以下的程序语句创建三个不同类型的边界：

```
Point3d lower = new Point3d(0, 0, 0); //立方体边界的下角
Point3d upper = new Point3d(1.0, 0.5, 1.5); //立方体边界的上角
BoundingBox box = new BoundingBox(lower, upper); //立方体边界
BoundingSphere sphere = new BoundingSphere(lower, 2); //球形边界
BoundingPolytope polytope = new BoundingPolytope(); //多面体边界
```

BoundingLeaf类对象则属于场景图中树的叶节点，并且在场景的局部坐标系中有确定的坐标。每个BoundingLeaf类对象都用一个Bounds类对象来定义其区域边界，此区域边界是相对于这个BoundingLeaf节点进行定位的。如果虚拟世界中有多个处于不同位置的节点需要一个共同的作用范围边界，则选用BoundingLeaf节点会更为简单。

我们来考虑一个给光源对象设置作用范围的例子，这个例子中，分别示例了两种设置作用范围边界的例子。

```
//直接设置作用范围边界
BoundingSphere bounds = new BoundingSphere(); //一个跳动的球
light.setInfluencingBounds(bounds);
//通过引用BoundingLeaf对象来设置作用范围边界
BoundingSphere bounds = new BoundingSphere(); //一个跳动的球
BoundingLeaf leaf = new BoundingLeaf(bounds);
root.addChild(leaf); //加到场景图
light.setInfluencingBoundingLeaf(leaf);
```

第一种方法创建了一个单位圆BoundingSphere对象，并直接把一个光源对象的作用范围设置为这个BoundingSphere对象。这个球体形状的作用范围的中心，位于光源对象的局部坐标系的坐标原点。

第二种方法也创建了一个类似的BoundingSphere对象，但是它是由一个BoundingLeaf节点进行引用的，将光源对象的作用范围设置到BoundingLeaf节点上。在这里，球形作用范围的中心位于BoundingLeaf叶节点所用局部坐标系的坐标原点，而BoundingLeaf叶节点的局部坐标系与光源对象的局部坐标系并不重合。如果场景图中有多个经过不同几何变换的光源，引用了同一个Bounds对象或同一个BoundingLeaf对象，那么Bounds类和BoundingLeaf类之间的区别会更为明显。如果这多个光源都直接引用了同一个Bounds对象，则实际定义各个光源的作用范围边界是各不相同的。如果它们引用的是BoundingLeaf对象，则实际上给各个光源定义了相同的作用范围边界。

程序清单5-4示例了Bounds对象对光源的作用效果。这个程序创建的场景中，有3个球体和一个光源，显示效果如图5-17所示。当这些球体位于光源的作用范围内时，它们会受到光源的光照。当用户用鼠标点击窗口面板时，程序会把光源的作用范围切换到三个作用范围中的下一个。第一次点击会缩小光源的作用范围，使得作用范围避开左侧球体。第二次点击将进一步缩小光源的作用范围，使其仅包含右侧的球体。第三次点击将恢复光源的初始作用范围。

157

程序清单5-4 TestBounds.java

```
1 package chapter5;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.media.j3d.*;
6 import javax.vecmath.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.awt.image.*;
10 import java.io.*;
11 import java.net.URL;
12 import javax.imageio.*;
13 import java.applet.*;
14 import com.sun.j3d.utils.applet.MainFrame;
15 //定义TestBounds类,继承自Applet类,演示Bounds对象在光照下的效果
16 public class TestBounds extends Applet {
17     public static void main(String[] args) {
18         new MainFrame(new TestBounds(), 640, 480); //创建主窗口并设置大小
19     }
20
21     Light light = null;
22     Bounds[] bounds = new Bounds[3];
23     int bIndex = 0;
24     //重写初始化方法
25     public void init() {
26         //创建Canvas3D画布对象
27         GraphicsConfiguration gc =
28             SimpleUniverse.getPreferredConfiguration();
29         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
30         setLayout(new BorderLayout()); //设置布局管理器
31         add(cv, BorderLayout.CENTER);
32         cv.addMouseListener(new MouseAdapter() { //添加鼠标事件侦听器
33             //点击鼠标时,修改光源的作用范围边界
34             public void mouseClicked(MouseEvent ev) {
35                 bIndex = (bIndex+1) % 3;
36                 System.out.println(bIndex);
37                 light.setInfluencingBounds(bounds[bIndex]);
38             }
39         });
40         BranchGroup bg = createSceneGraph(); //创建场景图分支
41         bg.compile(); //编译场景图
42         SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse对象
43         su.getViewingPlatform().setNominalViewingTransform();
44         su.addBranchGraph(bg);
45     }
46     //生成BranchGroup对象的私有方法,构造场景图
47     private BranchGroup createSceneGraph() {
48         BranchGroup root = new BranchGroup();
49         //第一个球体
50         Sphere sphere = new Sphere();
51         Transform3D tr = new Transform3D();
52         tr.setScale(0.1); //缩放变换
```



```
53 TransformGroup tg = new TransformGroup(tr);
54 root.addChild(tg);
55 tg.addChild(sphere);
56 //第二个球体
57 sphere = new Sphere();
58 tr.setTranslation(new Vector3f(-0.4f, 0f, 0f)); //平移变换
59 tg = new TransformGroup(tr);
60 root.addChild(tg);
61 tg.addChild(sphere);
62 //第三个球体
63 sphere = new Sphere();
64 tr.setTranslation(new Vector3f(-0.8f, 0f, 0f)); //平移变换
65 tg = new TransformGroup(tr);
66 root.addChild(tg);
67 tg.addChild(sphere);
68 //设置光源
69 light = new PointLight(new Color3f(Color.white),
70 new Point3f(1f,1f,1f),
71 new Point3f(1f,0f,0f)); //创建白色点光源, 允许动态修改光源属性
72 light.setCapability(Light.ALLOW_INFLUENCING_BOUNDS_WRITE);
73 //作用范围边界
74 bounds[0] = new BoundingSphere(new Point3d(0,0,0), 1);
75 bounds[1] = new BoundingSphere(new Point3d(0,0,0), 0.6);
76 bounds[2] = new BoundingSphere(new Point3d(0,0,0), 0.2);
77 light.setInfluencingBounds(bounds[0]);
78 root.addChild(light);
79 //设置背景
80 URL url = getClass().getClassLoader().getResource
81 ("images/bg.jpg");
82 BufferedImage bi = null;
83 try {
84 bi = ImageIO.read(url); //读入图像
85 } catch (IOException ex) {
86 ex.printStackTrace();
87 }
88 ImageComponent2D image =
89 new ImageComponent2D(ImageComponent2D.FORMAT_RGB, bi);
90 Background background = new Background(image); //背景节点
91 background.setApplicationBounds(bounds[0]);
92 root.addChild(background);
93 return root;
94 }
95 }
```

158

图5-18是这个程序的经过简化的场景图, 场景图中加入了三个Sphere对象。每个球体对象都各自附属到一个TransformGroup节点, 以对此球体进行缩放变换, 并移动到它们各自在场景中的位置。每个球体的位移量各不相同, 这样它们之间就不会相互重叠了(第49~67行)。

159

坐标(1, 1, 1)处放置了一个光源对象, 用于为球体对象提供光照。光源对象的作用范围边界决定了哪些图形对象会接收此光源的光照。程序的第74到76行创建了三个不同的BoundingSphere类对象, 它们的半径分别是1、0.6和0.2, 并将其记录到一个Bounds数组中。程序中定义了整数变量bIndex, 以指定由其中哪一个BoundingSphere对象作为光源对象的当前作用范围边界。初始时, 光源的作用范围边界为bounds[0], 该作用范围足以包含所有三个球体对象。

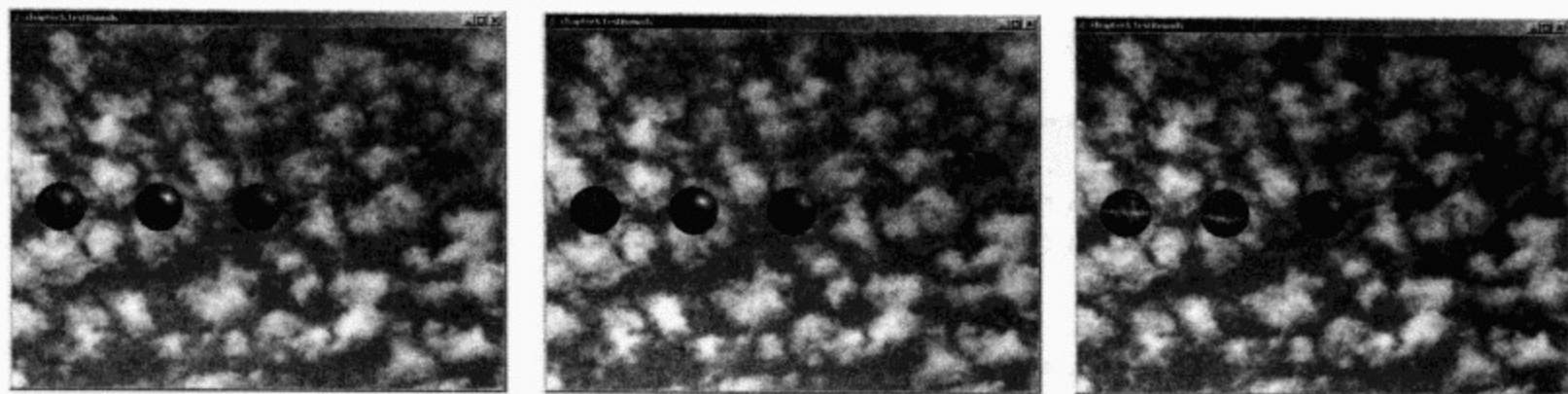


图5-17 作用范围的效果。左图：光源的作用范围包含了所有三个球体。中图：光源的作用范围只包含两个球体。右图：光源的作用范围进一步缩小，只包含一个球体

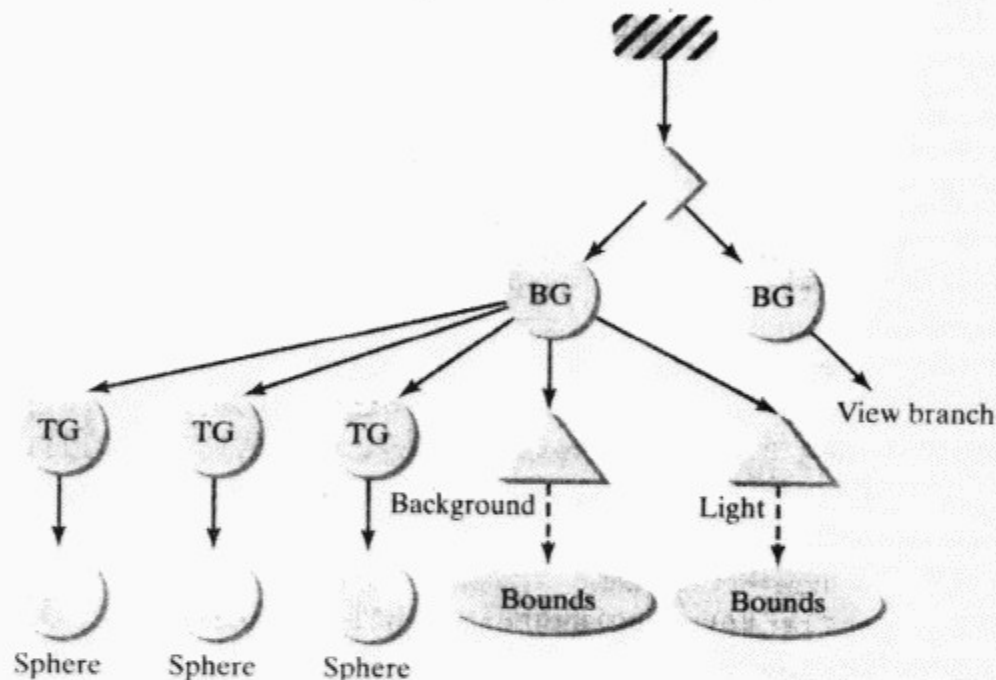


图5-18 程序清单5-4的场景图

程序中向Canvas3D对象添加了一个MouseListener类对象，以响应鼠标事件。当鼠标点击事件发生时，会改变作为光源对象的作用范围边界对象的索引：

```
bIndex = (bIndex+1) % 3;
light.setInfluencingBounds(bounds[bIndex]);
```

因此，程序在响应鼠标点击事件时，会以循环方式在bounds数组中进行选择。当bounds[1]被选中时，左侧的球体在光源对象的作用范围边界之外，因此变成了黑色。当bounds[2]被选中时，只有右侧的球体还在光源对象的作用范围边界内，其他两个球体都变成了黑色。

程序中用一个Background类叶节点定义了场景背景，场景背景被设置为一幅从文件载入的图像。Background节点同样需要用一个Bounds对象来限制其作用范围。在这个例子中，变量bounds[0]所指的BoundingSphere对象被用做背景节点的作用范围边界：

```
background.setApplicationBounds(bounds[0]);
```

5.10 场景图编译和能力位

在把一个以BranchGroup节点为根的场景图分支附属到Locale节点上并成为活动分支之前，可以对其进行编译。BranchGroup类有一个compile方法，可以用来对其代表的场景图分支进行编译（compiling）。场景图的编译会把场景图转化成一种内部表示，使得Java 3D绘制引擎的绘制效率更高。场景图的编译过程也让Java 3D有机会做一些优化以进一步加速绘制过程，但这种优化并未在Java 3D标准中定义，因而可能依赖于具体的实现。

一旦某个场景图成为活动场景图，虽然可以对其进行修改，但是每一个修改操作都必须显式地获得许可。这种许可由场景图节点或节点组件对象的能力位（capability bits）来定义。每个对象的每一种修改操作，都对应一个独立的能力位。默认情况下，所有的能力位都被设置为关闭状态，目的是提高绘制的效率。如果要对一个活动场景图的任何部分进行动态的修改，必须事先打开其对应的能力位。对能力位的设置必须在调用场景图的compile方法进行编译之前，如果程序运行时在没有设置相关能力位的情况下进行了修改操作，系统会抛出运行时异常。

160

可以通过调用SceneGraphObject类对象的下面这个方法来说能力位：

```
public final void setCapability(int bit);
```

每个节点类都定义了用于设置能力位的常量。如果需要设置多个能力位，就有必要多次调用setCapability方法，每次调用只能设置一个能力位。不能组合多个能力位，并通过一次调用进行设置。例如，如果要允许修改一个ColorintAttributes对象的颜色，需要设置相应的能力位：

```
coloring.setCapability(ColorAttributes.ALLOW_COLOR_WRITE);
```

如果要读取一个活动场景图的颜色，也需要获得相应的读许可：

```
coloring.setCapability(ColorAttributes.ALLOW_COLOR_READ);
```

如果对颜色属性既要进行读取又要进行修改，就需要分别设置两个能力位：

```
coloring.setCapability(ColorAttributes.ALLOW_COLOR_READ);
```

```
coloring.setCapability(ColorAttributes.ALLOW_COLOR_WRITE);
```

如果要修改一个Shape3D节点的几何属性，必须设置相应的能力位：

```
shape.setCapability(Shape3D.ALLOW_GEOMETRY_WRITE);
```

程序清单5-5示例了如何设置能力位，并对一个活动场景图的背景进行修改。这个程序示范了如何对活动场景图的属性进行修改，程序响应鼠标的点击，修改了背景图像和颜色（如图5-19所示）。

程序清单5-5 ChangeBackground.java

```
1 package chapter5;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.media.j3d.*;
6 import javax.vecmath.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.awt.image.*;
10 import java.io.*;
11 import java.net.URL;
12 import javax.imageio.*;
13 import java.applet.*;
14 import com.sun.j3d.utils.applet.MainFrame;
15 //定义ChangeBackground类，继承自Applet类，演示背景的改变
16 public class ChangeBackground extends Applet {
17     public static void main(String[] args) {
18         new MainFrame(new ChangeBackground(), 640, 480); //创建主窗口并设置大小
19     }
20
21     Background background = null;
22     ImageComponent2D image = null;
```

161


```

23 //重写初始化方法
24 public void init() {
25     GraphicsConfiguration gc =
26         SimpleUniverse.getPreferredConfiguration();
27     Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
28     setLayout(new BorderLayout()); //设置布局管理器
29     add(cv, BorderLayout.CENTER);
30     cv.addMouseListener(new MouseAdapter() { //添加鼠标事件侦听器
31         //点击鼠标时, 修改背景颜色和图像
32         public void mouseClicked(MouseEvent ev) {
33             if (background.getImage() == null)
34                 background.setImage(image);
35             else {
36                 background.setImage(null);
37                 float r = (float) Math.random();
38                 float g = (float) Math.random();
39                 float b = (float) Math.random();
40                 background.setColor(r, g, b);
41             }
42         }
43     });
44     BranchGroup bg = createSceneGraph(); //创建场景图分支
45     bg.compile(); //编译场景图
46     SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse对象
47     su.getViewingPlatform().setNominalViewingTransform();
48     su.addBranchGraph(bg);
49 }
50 //生成BranchGroup的私有方法, 构造场景图
51 private BranchGroup createSceneGraph() {
52     BranchGroup root = new BranchGroup();
53     //构造图形对象
54     Appearance ap = new Appearance(); //外观对象
55     ap.setMaterial(new Material()); //材质属性
56     Font3D font = new Font3D(new Font("SansSerif", Font.PLAIN, 1),
57         new FontExtrusion()); //Font3D节点组件
58     Text3D text = new Text3D(font, "Hello 3D"); //Text3D节点组件
59     Shape3D shape = new Shape3D(text, ap); //Shape3D叶节点
60     //几何变换
61     Transform3D tr = new Transform3D();
62     tr.setScale(0.5); //缩放变换
63     tr.setTranslation(new Vector3f(-0.95f, -0.2f, 0f)); //平移变换
64     TransformGroup tg = new TransformGroup(tr);
65     root.addChild(tg);
66     tg.addChild(shape);
67     //设置光源
68     PointLight light = new PointLight(new Color3f(Color.white),
69         new Point3f(1f, 1f, 1f),
70         new Point3f(1f, 0.1f, 0f)); //添加白色点光源
71     BoundingSphere bounds = new BoundingSphere(); //球体作用范围边界对象
72     light.setInfluencingBounds(bounds);
73     root.addChild(light);
74     //初始化背景节点
75     background = new Background(1.0f, 1.0f, 1.0f);
76     background.setApplicationBounds(bounds); //设置作用范围边界

```



```
77    //读入图像
78    URL url = getClass().getClassLoader().
79    getResource("images/bg.jpg");
80    BufferedImage bi = null;
81    try {
82        bi = ImageIO.read(url);
83    } catch (IOException ex) {
84        ex.printStackTrace();
85    }
86    image = new ImageComponent2D(ImageComponent2D.FORMAT_RGB, bi);
87    //允许动态修改颜色和图像属性
88    background.setCapability(Background.ALLOW_COLOR_WRITE);
89    background.setCapability(Background.ALLOW_IMAGE_READ);
90    background.setCapability(Background.ALLOW_IMAGE_WRITE);
91    root.addChild(background);
92    return root;
93 }
94 }
```

162

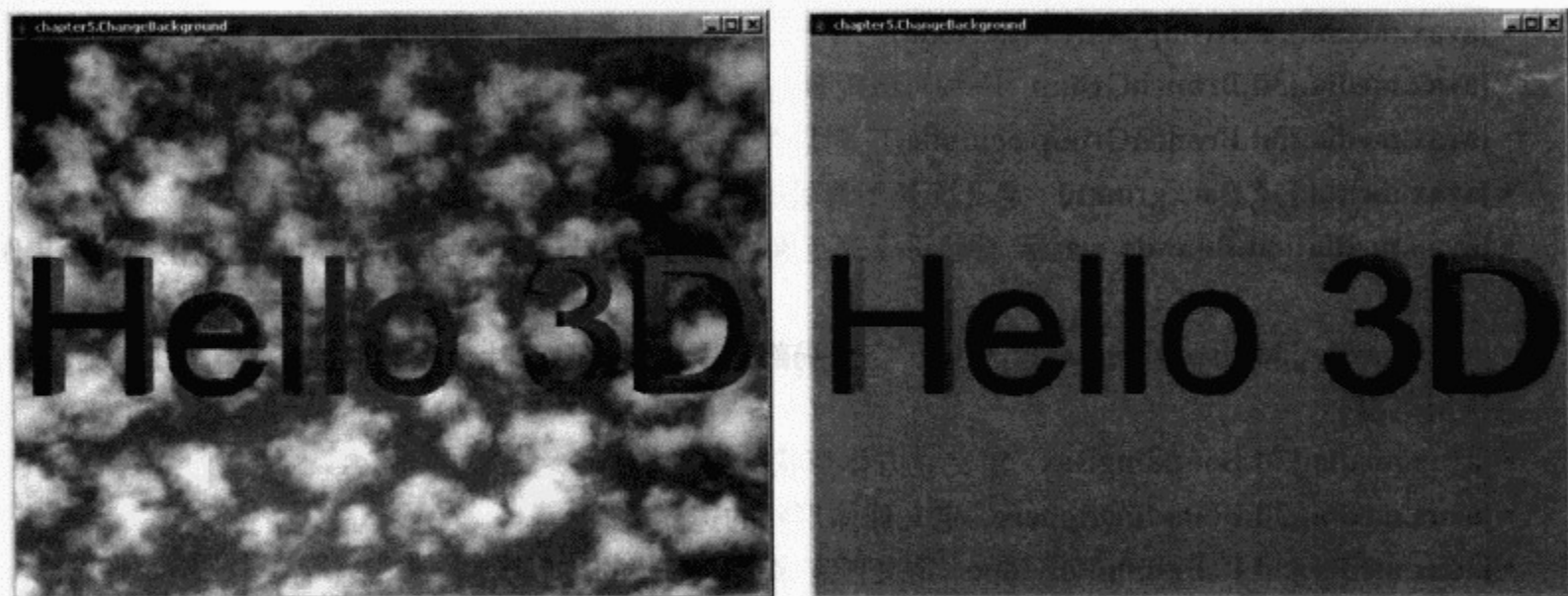


图5-19 天空图像背景与随机单色背景之间的切换

这个例子与程序清单5-2相似，它们的场景图完全相同。这个例子中使用了两种背景：图像的背景和单色的背景。

程序从文件“bg.jpg”读入图像，并保存在一个BufferedImage类对象中，此对象随后被传递给一个ImageComponent2D类对象（第77到84行）。程序中设置了Background类对象的能力位，以便在活动状态下对其颜色和图像属性进行修改，以及对其图像属性进行读取（第88到90行）。

```
ALLOW_COLOR_WRITE
ALLOW_IMAGE_READ
ALLOW_IMAGE_WRITE
```

在Canvas3D对象的构造函数中，设置了此Canvas3D对象要响应鼠标事件。在鼠标点击事件响应函数mouseClicked中，程序检查背景，判断是否存在背景图像。如果背景节点中不包含图像，则把载入的图像赋予背景节点。如果背景中已经有图像存在，则把背景的图像设为空，并把背景节点的颜色属性设为一个随机颜色。因此，连续地点击鼠标，将使得场景的背景在图像和随机单色之间切换。

主要的类和方法

163

- **com.sun.j3d.util.applet.MainFrame** 在窗口内显示Java小程序 (applet) 的功能类。
- **javax.media.j3d.VirtualUniverse** 包装了整个虚拟宇宙的3D空间坐标系的类。
- **javax.media.j3d.Locale** 用于定义锚定于虚拟宇宙中某个位置的局部坐标空间, 其局部坐标为float数据类型。
- **javax.media.j3d.HiResCoord** 一个定点型数据类型, 用于表示虚拟宇宙中的高精度坐标。
- **com.sun.j3d.util.universe.SimpleUniverse** 一个方便程序编写的类, 包含了虚拟宇宙 (VirtualUniverse类) 的一个默认实现、一个Locale对象和场景图的一个view分支。
- **javax.media.j3d.SceneGraphObject** 一个抽象类, 用做所有场景图元素类的最顶层基类。
- **javax.media.j3d.Node** 场景图中的节点类。
- **javax.media.j3d.NodeComponent** 场景图中的节点组件类。
- **javax.media.j3d.Group** 场景图中的组节点类。
- **javax.media.j3d.Leaf** 场景图中的叶节点类。
- **javax.media.j3d.SceneGraphObject.setCapability(int)** 用于允许对活动场景图中的某些对象进行特定操作的方法。
- **javax.media.j3d.Group.addChild(Node)** 用于向场景图中添加子节点的方法。
- **javax.media.j3d.BranchGroup** 一种可以附属到场景图的Locale对象的特殊组节点。
- **javax.media.j3d.BranchGroup.compile()** 对场景图进行编译以提高性能的方法。
- **javax.media.j3d.Background** 定义场景图背景的颜色、图像和几何属性的叶节点。
- **javax.media.j3d.Bounds** 封装了空间局部范围边界信息的节点组件类, 被环境节点用来限制其作用范围。
- **javax.media.j3d.BoundingLeaf** 封装了空间局部范围边界信息的叶节点类, 被环境节点用来限制其作用范围。
- **javax.media.j3d.BoundingBox** 定义立方体形状边界类。
- **javax.media.j3d.BoundingSphere** 定义球体形状边界类。
- **javax.media.j3d.BoundingPolytope** 定义任意多面体形状边界类。

关键术语

- **几何特征 (geometry)** 可视对象的结构性定义。
- **外观 (appearance)** 可视对象的一组绘制属性。
- **有向无环图 (directed acyclic graph, DAG)** 不包含环路的有向图。
- **场景图 (scene graph)** 一个用于定义需要绘制的图形场景的有向无环图。
- **树 (tree)** 一种以递归方式加入不同子节点的图结构。
- **叶节点 (leaf node)** 树结构中没有子节点的节点。
- **内部节点 (internal node)** 树结构中有子节点的节点。
- **基元 (primitive)** 用于图形物体建模的基本可视对象。
- **能力位 (Capability bit)** SceneGraphObject类对象内的一种标志位, 用于决定是否允许对一个活动场景图进行某种特定的操作。

本章提要

- 本章讨论了3D计算机图形的基本概念和Java 3D系统的基本架构。
- 在3D图形系统中, 我们构造一个虚拟世界来对3D图形场景进行建模, 从某个视角对此模型进行观察, 最终得到此场景的绘制结果。

164

- Java 3D API是基于场景图这一关键概念构造的，场景图把所需绘制的图形场景的所有图形对象的描述和属性，都纳入到一个单一的数据结构中。
- 介绍了构造场景图及其构建模块的规则。每个场景图都是一个有向无环图，它的每一个节点都是一个超结构类、节点类或节点组件类的实例对象。
- 介绍了Java 3D程序的总体结构。我们只需用场景图及其相关对象构造一个3D图形场景的模型，就可以由Java 3D引擎来自动绘制整个场景。
- 可以用Background类叶节点来改变场景背景，诸如Background类节点和Light类节点之类的环境类节点，需要设置边界以限制其对绘制过程的作用范围。Bounds类和BoundingLeaf类对象，是设置作用范围边界的两种方式。
- 对场景图分支进行编译，可以提高绘制效率。只有在设置了相关的能力位后，才可以对活动场景图中的组件进行修改。

复习题

- 5.1 如果一棵树有15个节点，请问它共有多少条边？
- 5.2 下面这个图（图5-20）是不是一棵树？是不是一个有向无环图？如果它是一棵树，请指出它的根、叶节点和内部节点。
- 5.3 下面这个图（图5-21）是不是一棵树？是不是一个有向无环图？如果它是一棵树，请指出它的根、叶节点和内部节点。
- 5.4 下面这个图（图5-22）是不是一棵树？是不是一个有向无环图？如果它是一棵树，请指出它的根、叶节点和内部节点。

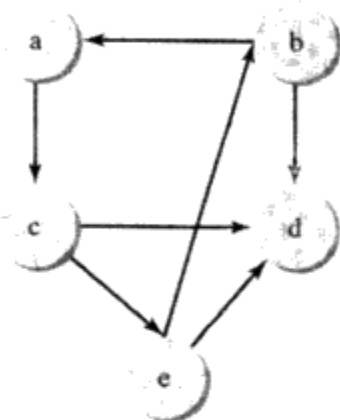


图5-20 5.2题图

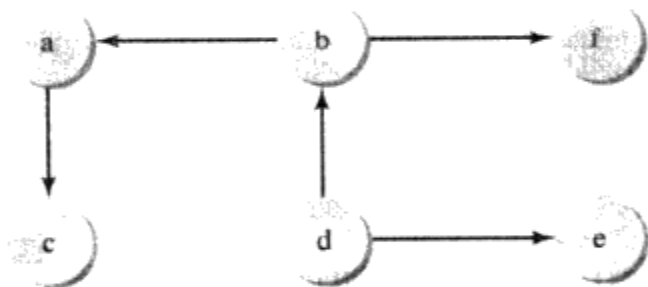


图5-21 5.3题图

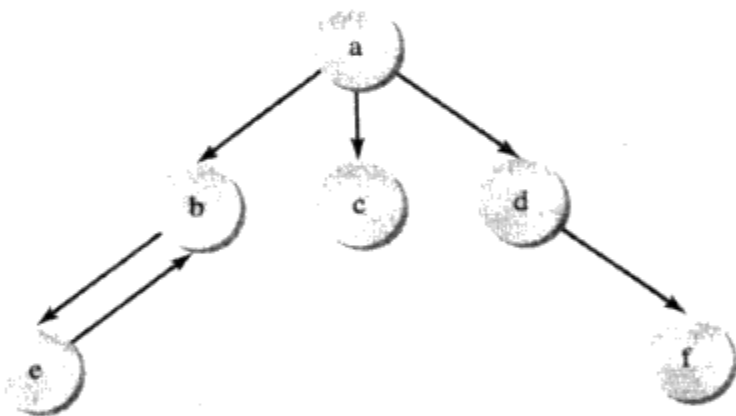


图5-22 5.4题图

- 5.5 画出以下Java 3D代码段对应的场景图：

```
BranchGroup root = new BranchGroup();
TransformGroup trans = new TransformGroup();
root.addChild(trans);
Shape3D shape1 = new Shape3D();
Shape3D shape2 = new Shape3D();
Shape3D shape3 = new Shape3D();
trans.addChild(shape1);
trans.addChild(shape2);
root.addChild(shape3);
```

- 5.6 画出以下Java 3D代码段对应的场景图：

```
BranchGroup root = new BranchGroup();
TransformGroup trans1 = new TransformGroup();
TransformGroup trans2 = new TransformGroup();
root.addChild(trans1);
```



```

root.addChild(trans2);
Light light = new PointLight();
trans1.addChild(light);
Switch switch = new Switch();
trans2.addChild(switch);
Shape3D shape1 = new Shape3D();
Shape3D shape2 = new Shape3D();
switch.addChild(shape1);
switch.addChild(shape2);
Appearance appear = new Appearance();
shape1.setAppearance(appear);
shape2.setAppearance(appear);

```

166 5.7 请编写与图5-23中的场景图分支对应的Java 3D代码段。

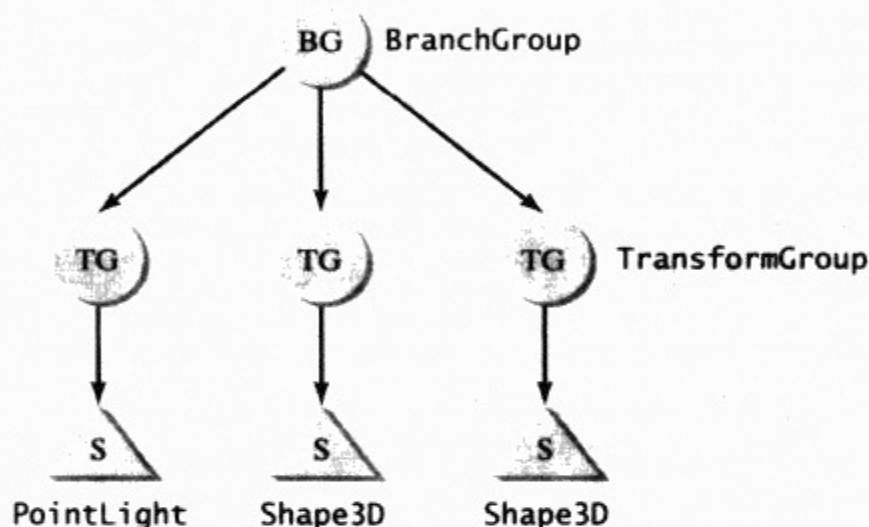


图5-23 5.7题的场景图

5.8 请编写与图5-24中的场景图分支对应的Java 3D代码段。

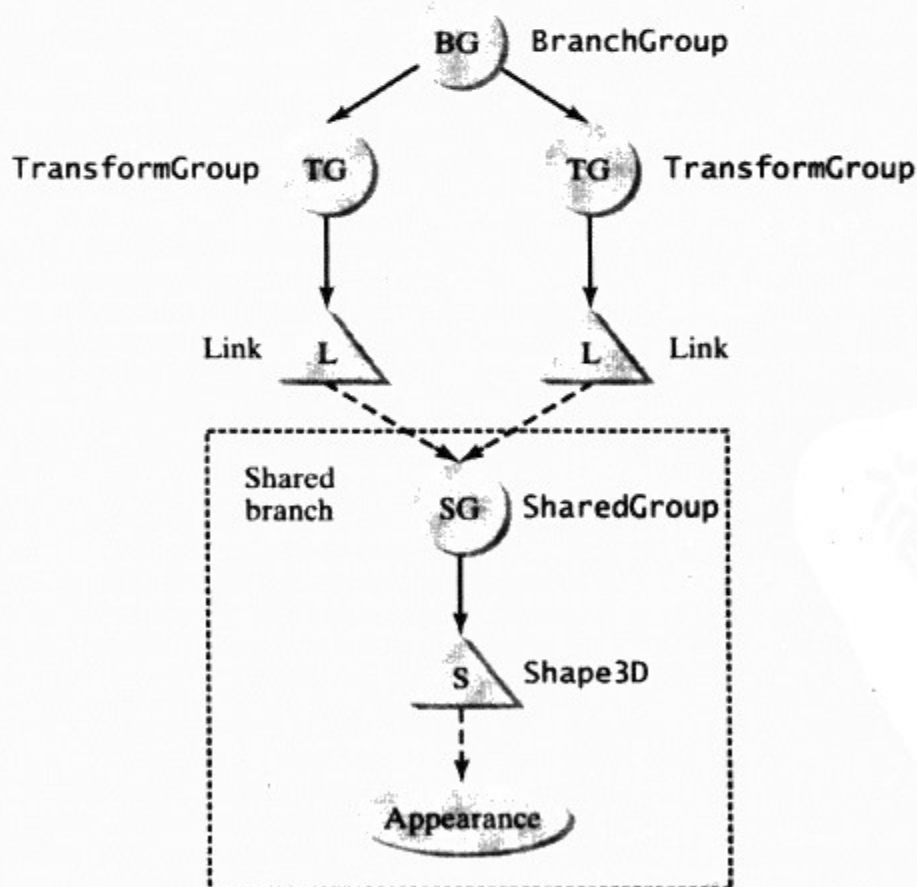


图5-24 5.8题的场景图

编程练习

- 5.1 ColorCube类是Shape3D类的子类，可以用做表示带彩色面的立方体对象的叶节点。请编写一个Java 3D程序，用SimpleUniverse类来显示一个ColorCube对象。
- 5.2 和第5.1题一样，编写一个显示一个ColorCube类对象的Java 3D程序，但不使用SimpleUniverse类。画出该程序的完整场景图。 167
- 5.3 用Background类叶节点向第5.1题的程序中加入一个蓝色背景对象，用一个球体边界对象直接设置此背景对象的作用范围。
- 5.4 编写一个类似于5.3题的Java 3D程序，但要求用BoundingLeaf类节点来把背景对象的作用范围边界设置为一个立方体边界。
- 5.5 编写一个类似于程序清单5-4的Java 3D程序，但要求用BoundingLeaf类节点来设置光源对象的作用范围边界。 168

第6章 图形内容

学习目标

- 理解可视对象的两个基本属性：几何特征和外观。
- 描述点和向量的表示。
- 使用GeometryArray类族构造几何体。
- 使用GeometryInfo类构造几何体。
- 使用几何基元。
- 将文本和字体作为几何对象使用。
- 使用Appearance类以及相关的节点组件类。

169

6.1 引言

可视形体是3D图形模型的基本图形构造块 (graphics building block)，它们构成了绘制场景中的可视对象。一个形状对象通常由其几何属性和外观属性定义。几何属性提供了对象的形状、大小以及其他结构属性的数学描述；外观属性则定义了物体的颜色、纹理、材质以及其他一些与外观相关的特性。

可视对象的几何体通常由一系列简单形体（如三角形）构成，也可以预定义一些更为复杂的可复用的形体（如立方体和球体），称为几何基元 (primitives)。几何基元提供了一层抽象，使用几何基元可以简化复杂对象的构造过程。基于字体的文本提供了另一种几何形体，2D和3D文本对象都是构造3D场景的重要元素。

Java 3D中提供了叶节点类Shape3D来表示各种形体。Shape3D节点通过引用Geometry类和Appearance类对象来定义其几何属性和外观属性。Geometry类派生出多个派生类，以使用不同的方式定义不同类型的几何形体，Appearance类则通过引用各种属性对象来定义形体外观的各方面属性。Java 3D中提供了一些常用的图形基元，如长方体 (box)、球体 (sphere)、圆柱体 (cylinder)、圆锥体 (cone) 等。Java 3D也支持将2D文本和3D文本作为几何形体进行处理。

在本章中，我们将介绍使用几何属性和外观属性的定义来构造可视对象，考察使用Java 3D构造低层次几何形体的功能。本章也介绍了使用几何基元和文本对象来构造高层几何形体，我们还将讨论Java 3D外观属性描述的基本结构和用法。涉及更高级绘制选项（如光照和纹理）的外观属性，将在后续章节中进行更详细的讨论。

6.2 点和向量

要对几何体进行建模，首先需要对点进行建模。为了在计算机中精确表示点，通常引入代数学中的向量 (vector) 和向量空间 (vector space) 的概念，将一个 n 维的向量表示为一个数值的 n 元组：

$$(x_1, x_2, \dots, x_n)$$

所有 n 维向量组成的集合构成了 n 维空间 R^n 。在3D空间中，一个点可以表示为一个3D向量 (x, y, z) 。在齐次坐标系下，每个点与一个四维向量 (x, y, z, w) 相关联。

此外，还有几何概念下的向量，表示带有方向的量，这样的例子包括直线的方向、力、速率和加速度等。一个几何向量也表示为一个 n 元组，从代数角度看，几何点和几何向量之间并没有区别，只在一般性的数学量的解释上存在差异。

在3D图形学中，几何体的构造和变换在很大程度上依赖于向量的数学表示。Java语言在javax.vecmath包中提供了丰富的类用于表示点、向量和矩阵。Java 3D中的类会频繁使用到这些向量代数类，javax.vecmath包包含在Java 3D的发布版本中。

javax.vecmath包中包含了多种向量和矩阵类。图6-1列出了其中的部分向量类。

类名的后缀表明了向量的维度及各分量的数据类型，数据类型后缀列表如下：

- f: float类型。
- d: double类型。
- i: int类型。
- b: byte类型。

例如，“4d”表示这是一个四维的double类型向量。类名的词干则表明了这些类的用途类别。

- Tuple*类：数据元组的抽象基类。
- Color*类：颜色表示。
- Point*类：几何点。
- Vector*类：几何向量。
- TexCoord*类：纹理映射坐标。
- Quat*类：四元组。

除了数据表示，向量类还包括了相应类别的标准操作方法，如Tuple4f类包含了如下的方法：

- void add(Tuple4f t1)：加另一个元组。
- void sub(Tuple4f t1)：减另一个元组。
- void scale(float k)：缩放元组。
- void negate()：对每个分量取反。

Point3D类包含了一些与距离计算相关的方法：

- void distance(Point3D p1)：求到另一个点的距离。
- void distanceL1(Point3D p1)：求到另一个点的 L^1 距离。
- void distanceLinf(Point3D p1)：求到另一个点的 L^∞ 距离。

Vector3D类中，包含了如下一些向量操作方法：

- double dot(Vector3D v1)：计算该向量与另一向量的内积（向量点乘运算）。
- double cross(Vector3D v1, Vector3D v2)：计算两个向量的外积（向量叉乘运算）。
- double length()：计算该向量的长度（模）。
- double angle(Vector3D v1)：计算该向量与另一给定向量的夹角。

向量代数对象很容易创建和操作，下面的实例代码创建了两个Point3D对象，并计算两者之

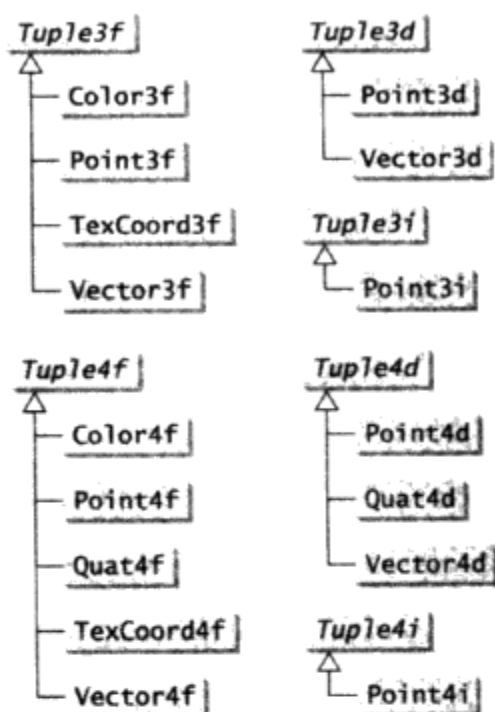


图6-1 向量数学类

170

171

间的距离：

```
Point3d p1 = new Point3d(1.0, 2.3, 0.0);
Point3d p2 = new Point3d(0.0, -0.5, 1.2);
double dist = p1.distance(p2);
```

我们还可以调用angle方法来计算两个向量之间的夹角：

```
Vector3f v1 = new Vector3f(1.0, 2.3, 0.0);
Vector3f v2 = new Vector3f(0.0, -0.5, 1.2);
double angle = v1.angle(v2);
```

6.3 几何特征

3D对象的基本几何体通常用点 (point)、线 (line) 和曲面 (surface) 来建模。点和线 (包括曲线) 的定义相对比较简单, 因为它们通常可由对应的2D模型直接扩展得到, 曲面模型则提出了真正的挑战。3D实体通常由一组曲面进行建模。在数学上, 一个曲面通常用坐标系上的隐式方程 (implicit equation) 表示:

$$F(x, y, z) = 0$$

但是在图形学领域中, 通常采用等价的更便于图形应用的参数方程 (parametric equation) 定义:

$$\begin{aligned} x &= f(u, v) \\ y &= g(u, v) \\ z &= h(u, v) \end{aligned}$$

由于表示一个任意3D曲面往往比较复杂, 我们通常使用一组简单曲面的集合来拟合该曲面。用简单多边形 (如三角形和四边形) 构成网格是较常用的一种表示方案, 另一种使用面广而强大的曲面表示工具, 是采用多项式和样条曲面。图6-2为使用多边形网格 (polygon meshes) 表示曲面的一个实例。

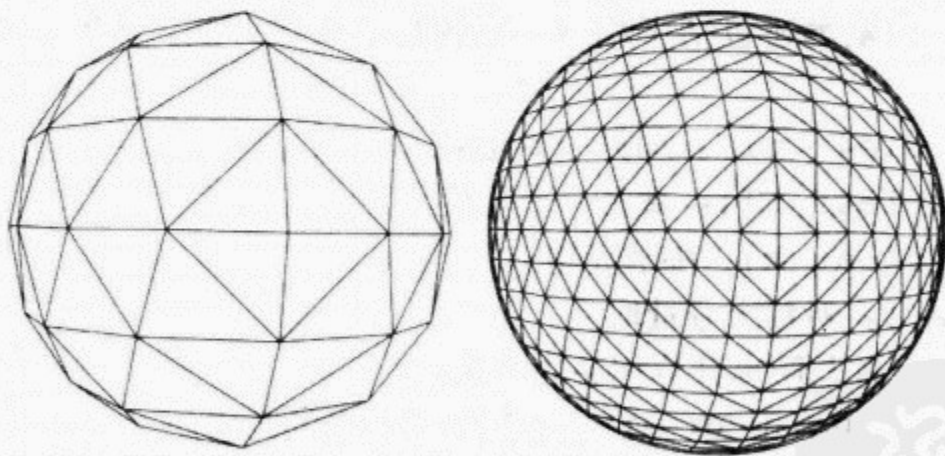


图6-2 一个球体在不同分辨率下的三角形网格表示

Java 3D支持直接使用基本的点、线、三角形或四边形数组构造几何体, 它也支持使用高层次的几何对象, 如几何基元和3D文本对象。

在一个Java 3D场景图中, 可视对象通常表示为一个Shape3D叶节点, 该Shape3D对象引用一个Geometry对象, 用于定义可视对象的形状和其他几何特征。Shape3D节点还引用了Appearance对象, 以定义其绘制时的外观。图6-3为一

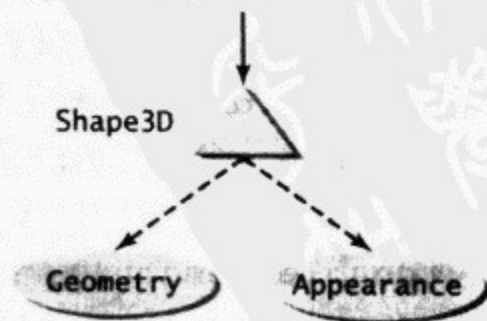


图6-3 一个典型的场景图形状节点

个典型的场景图中Shape3D节点的配置实例。

Geometry类是一个抽象类，它派生出大量派生类，对应的类层次结构如图6-4所示。

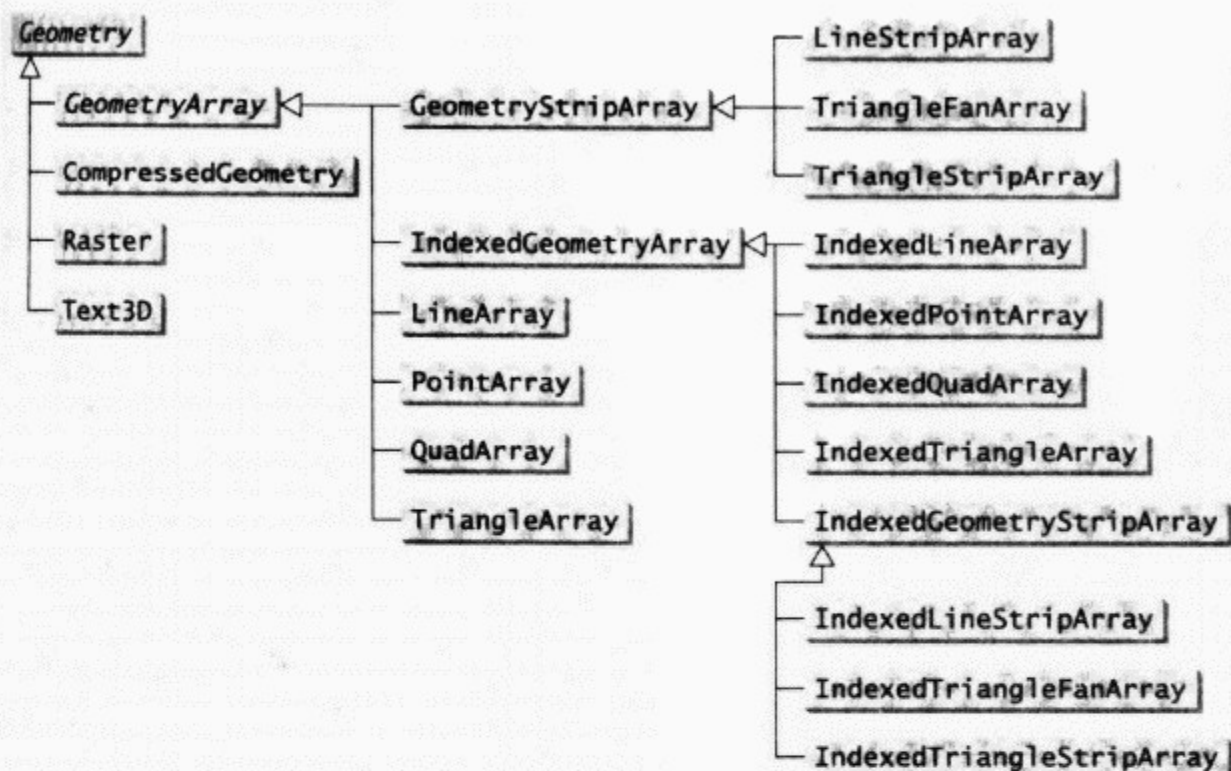


图6-4 Geometry类层次结构

173

6.3.1 类GeometryArray

GeometryArray类族支持用户直接使用简单多边形、线或点数组构造几何体。GeometryArray定义了图形的顶点，并指定了顶点间的结构关系。

在一个GeometryArray对象中，顶点的定义一般包含坐标值，此外还可能会包含一些其他类型的数据，如曲面上过该点的法向量和颜色。是否包含这些数据成分，由如下的位掩码指定。

COORDINATES：顶点坐标。

NORMALS：顶点法向量。

COLOR_3：不带alpha值的颜色。

COLOR_4：带alpha值的颜色。

TEXTURE_COORDINATE_2：2D纹理坐标。

TEXTURE_COORDINATE_3：3D纹理坐标。

TEXTURE_COORDINATE_4：四维纹理坐标。

这些位掩码可通过位或运算（OR，“|”）结合使用，掩码的设置可在GeometryArray的构造函数中进行。这些数据成分如果存在，则在所有的顶点中都会包含这些数据，顶点的颜色描述可用于确定可视对象在某些特定光照模型下的颜色。曲面法向量在光照模式下计算光反射时是必需的，纹理坐标定义了纹理空间中的坐标。光照模型和纹理映射将在第9章进行讨论。

GeometryArray类族的对象，可根据数据成分和数组大小，使用合适的构造函数创建得到。GeometryArray类提供了多种不同的方法，用于设置坐标值以及其他数据。例如，可以设定单个坐标值，也可以同时设定整个坐标数组：

```
void setCoordinate(int index, Point3f coord)
void setCoordinates(int startIndex, Point3f[] coords)
```

PointArray类定义了一个由一个点集构成的几何体，每个顶点的定义与图形中的一个点相

关联。例如，下面的代码段定义了一个包含三个点的PointArray图形，其几何体如图6-5所示。

```
PointArray pa = new PointArray(3, GeometryArray.COORDINATES); //创建点集对象
//以此设定坐标
pa.setCoordinate(0, new Point3f(0f, 0f, 0f));
pa.setCoordinate(1, new Point3f(1f, 0f, 0f));
pa.setCoordinate(2, new Point3f(0f, 1f, 0f));
```

LineArray类定义了线段的几何属性，每两个顺序指定的顶点确定了几何体中的一个线段：

```
174 LineArray la = new LineArray(6, GeometryArray.COORDINATES);
//设定线段端点数组
Point3f[] coords = new Point3f[6];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(1f, 1f, 0f);
coords[2] = new Point3f(1f, 0f, 0f);
coords[3] = new Point3f(2f, 1f, 0f);
coords[4] = new Point3f(2f, 1f, 0f);
coords[5] = new Point3f(3f, 0f, 0f);
la.setCoordinates(0, coords);
```

上述代码定义的几何体如图6-6所示。

TriangleArray类定义了由三角形片组成的面，每三个顶点定义了一个三角形。下面的代码段定义了一个由两个三角形组成的几何对象：

```
TriangleArray ta = new TriangleArray(6, GeometryArray.COORDINATES);
//设定三角形顶点数组
Point3f[] coords = new Point3f[6];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(1f, 1f, 0f);
coords[2] = new Point3f(1f, 0f, 0f);
coords[3] = new Point3f(1f, 0f, 0f);
coords[4] = new Point3f(2f, 1f, 0f);
coords[5] = new Point3f(3f, 0f, 0f);
ta.setCoordinates(0, coords);
```

由上述代码定义的图形如图6-7所示。

我们也可以通过TriangleArray使用一系列三角形来构造一个圆锥体：

```
int n = 60; //三角形片的数目
TriangleArray ta = new TriangleArray(3*n, GeometryArray.COORDINATES);
//创建三角形数组对象，顶点数目为180
Point3f apex = new Point3f(0, 0, 1); //圆锥顶点坐标
Point3f p1 = new Point3f(1, 0, 0);
int count = 0;
for (int ii = 1; ii <= n; ii++) {
//以此设置三角形片的顶点
float x = (float)Math.cos(ii*2*Math.PI/n);
float y = (float)Math.sin(ii*2*Math.PI/n);
Point3f p2 = new Point3f(x, y, 0);
ta.setCoordinate(count++, apex);
ta.setCoordinate(count++, p1);
ta.setCoordinate(count++, p2);
p1 = p2;
}
175 }
```

圆锥体的底面圆周被分成 n 段，每段的两个端点及圆锥顶点组成一个三角形。TriangleArray

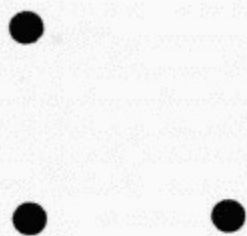


图6-5 一个由PointArray构成的几何体



图6-6 由LineArray定义的几何体

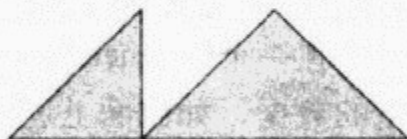


图6-7 由TriangleArray定义的几何体

需要显式指定每个三角形的每一个顶点，因此尽管实际上只有 $n+1$ 个不同的点，我们仍需要定义 $3n$ 个坐标值。

QuadArray定义了一个四边形片组成的曲面，每组连续的四个顶点定义了一个四边形，这四个顶点必须在同一个平面上。如下的QuadArray对象包含两个不在同一平面的正方形，但每一个正方形的顶点都在同一个平面上（如图6-8所示）。

```
QuadArray qa = new QuadArray(8, GeometryArray.COORDINATES);
//依次定义两个四边形顶点
Point3f[] coords = new Point3f[8];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(1f, 0f, 0f);
coords[2] = new Point3f(1f, 1f, 0f);
coords[3] = new Point3f(0f, 1f, 0f);
coords[4] = new Point3f(1f, 1f, 0f);
coords[5] = new Point3f(0f, 1f, 0f);
coords[6] = new Point3f(0f, 1f, 1f);
coords[7] = new Point3f(1f, 1f, 1f);
qa.setCoordinates(0, coords);
```

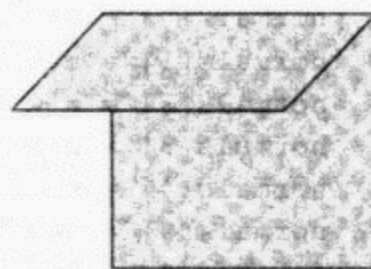


图6-8 由QuadArray定义的几何体

使用QuadArray对象必须指定每一个四边形的四个顶点，因此在这个图形中，尽管只有6个不同的点，我们仍然需要定义8个顶点。

除坐标值外，顶点的其他属性（如法向量和颜色）也可通过类似的方式进行设置。例如，如下的TriangleArray对象除了定义顶点坐标，也定义了顶点的颜色：

```
TriangleArray ta = new TriangleArray (6,
    GeometryArray.COORDINATES | GeometryArray.COLOR_3);
//同时定义坐标和颜色
//开始定义顶点坐标值
Point3f[] coords = new Point3f[6];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(1f, 1f, 0f);
coords[2] = new Point3f(1f, 0f, 0f);
coords[3] = new Point3f(1f, 0f, 0f);
coords[4] = new Point3f(2f, 1f, 0f);
coords[5] = new Point3f(3f, 0f, 0f);
ta.setCoordinates(0, coords); //设定顶点坐标数组
//开始定义颜色值
Color3f[] colors = new Color3f[6];
colors[0] = new Color3f(1f, 0f, 0f);
colors[1] = new Color3f(0f, 1f, 0f);
colors[2] = new Color3f(0f, 0f, 1f);
colors[3] = new Color3f(1f, 1f, 0f);
colors[4] = new Color3f(0f, 1f, 1f);
colors[5] = new Color3f(1f, 0f, 1f);
ta.setColors(0, colors); //设定顶点颜色数组
```

曲面法向量也可以在几何数组中指定，如下的QuadArray对象中，包含了法向量的描述：

```
QuadArray qa = new QuadArray(8,
    GeometryArray.COORDINATES | GeometryArray.NORMALS);
//同时定义坐标和法向量
//开始定义顶点坐标值
Point3f[] coords = new Point3f[8];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(1f, 0f, 0f);
```



```

coords[2] = new Point3f(1f, 1f, 0f);
coords[3] = new Point3f(0f, 1f, 0f);
coords[4] = new Point3f(1f, 1f, 0f);
coords[5] = new Point3f(0f, 1f, 0f);
coords[6] = new Point3f(0f, 1f, 1f);
coords[7] = new Point3f(1f, 1f, 1f);
qa.setCoordinates(0, coords); //设定顶点坐标
//开始定义顶点法向量
Vector3f[] normals = new Vector3f[8];
normals[0] = new Vector3f(0f, 0f, 1f);
normals[1] = new Vector3f(0f, 0f, 1f);
normals[2] = new Vector3f(0f, 0f, 1f);
normals[3] = new Vector3f(0f, 0f, 1f);
normals[4] = new Vector3f(0f, 1f, 0f);
normals[5] = new Vector3f(0f, 1f, 0f);
normals[6] = new Vector3f(0f, 1f, 0f);
normals[7] = new Vector3f(0f, 1f, 0f);
qa.setNormals(0, normals); //设定顶点法向量数组

```

6.3.2 类GeometryStripArray

数组中的顶点经常被多个多边形所共用，而使用TriangleArray或QuadArray，需要重复添加这些公用的顶点。我们可以采用两种方案来提高效率，一种是使用GeometryStripArray类，该类使用带（strip）的概念以允许共享相邻的顶点。为了定义不同的带，每个带中顶点的数目由一个整型数组指定：

```
void setStripVertexCounts(int[] stripVertexCounts);
```

数组的长度即为带的数目，数组每一个元素的数值，对应一个带的顶点数。

GeometryStripArray有三个子类，其中LineStripArray定义了折线带，该类使用一个点序列确定一个带，避免了重复内部点。例如，下面的代码使用LineStripArray对象，定义了与图6-6相同的几何体：

```

int[] stripVertexCounts = {2, 3}; //共有两个带，带的顶点数分别为2和3
LineStripArray lsa = new LineStripArray(5, GeometryArray.COORDINATES,
    stripVertexCounts); //创建LineStripArray对象，由5个不重复的顶点构成
//开始设定顶点坐标
Point3f[] coords = new Point3f[5];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(1f, 1f, 0f);
coords[2] = new Point3f(1f, 0f, 0f);
coords[3] = new Point3f(2f, 1f, 0f);
coords[4] = new Point3f(3f, 0f, 0f);
lsa.setCoordinates(0, coords); //设定LineStripArray对象的顶点坐标数组

```

TriangleStripArray类定义了三角形带，每一个带中的三个连续顶点定义了一个三角形。图177 6-9为一个由TriangleStripArray对象构造的几何体。

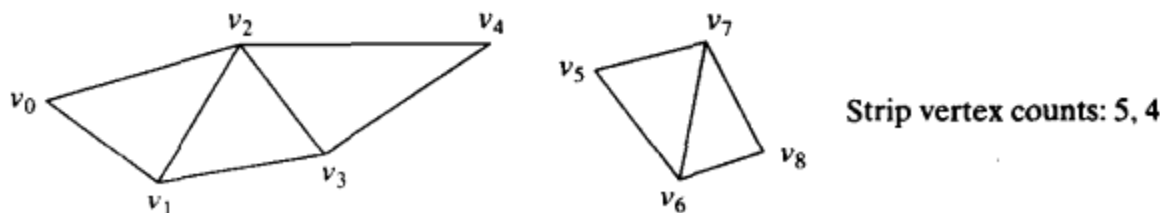


图6-9 一个TriangleStripArray

TriangleFanArray类定义了另一种三角形带，每一个带中的第一个点与其他的顶点中每两个连续的点构成一个三角形。图6-10为一个由TriangleFanArray对象构造的几何体实例。

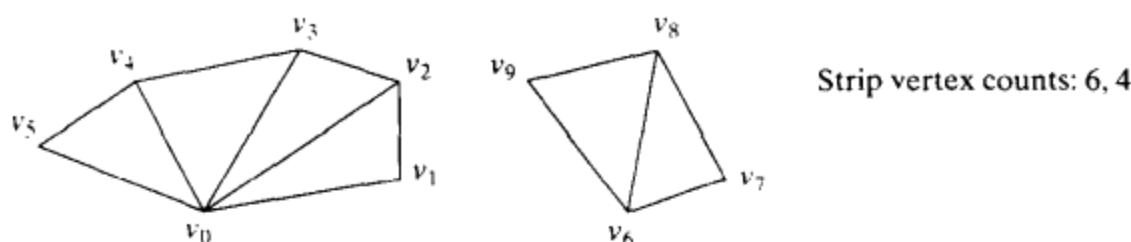


图6-10 一个TriangleFanArray

使用TriangleFanArray，可以很方便地定义出一个圆锥体的几何体：

```
int n = 60; //三角形片的数目
int[] stripVertexCounts = {n+2}; //只定义了一个带，带的点数目为62
TriangleFanArray tfa = new TriangleFanArray
(n+2, GeometryArray.COORDINATES, stripVertexCounts);
//创建TriangleFanArray实例，由62个点组成
Point3f apex = new Point3f(0, 0, 1);
tfa.setCoordinate(0, apex);
for (int ii = 0; ii <= n; ii++) {
    //以此设定带中的点
    float x = (float)Math.cos(ii*2*Math.PI/n);
    float y = (float)Math.sin(ii*2*Math.PI/n);
    Point3f p = new Point3f(x, y, 0);
    ta.setCoordinate(ii+1, p);
}
```

这里，我们只使用了一个包含 $n+2$ 个点的带，就定义了 n 个三角形片。

6.3.3 类IndexedGeometryArray

另一个解决重复顶点问题的方法是，使用IndexedGeometryArray类。与直接指定顶点来定义多边形不同的是，IndexedGeometryArray对象指定了多边形顶点在一个顶点数组中的索引值，从而使每一顶点只需要定义一次，但是可以被引用多次。例如，下面的IndexedQuadArray对象定义了一个由两个正方形组成的图形，其结果如图6-8所示。该图形只使用了6个顶点，而不是像GeometryArray所需要的8个顶点，每一个四边形由四个索引值所对应的顶点确定。

```
IndexedQuadArray iqa = new IndexedQuadArray
(6, GeometryArray.COORDINATES, 8); //创建IndexedQuadArray实例
//开始设顶点数组
Point3f[] coords = new Point3f[6];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(1f, 0f, 0f);
coords[2] = new Point3f(1f, 1f, 0f);
coords[3] = new Point3f(0f, 1f, 0f);
coords[4] = new Point3f(0f, 1f, 1f);
coords[5] = new Point3f(1f, 1f, 1f);
iqa.setCoordinates(0, coords);
//设置图形顶点索引数组
int[] indices = {0, 1, 2, 3, 2, 3, 4, 5};
iqa.setCoordinateIndices(0, indices);
```

其他的属性（如法向量和颜色）也可以通过类似的方法进行索引。

此外，还有IndexedGeometryStripArray类及其子类IndexedLineStripArray、Indexed-

TriangleStripArray和IndexedTriangleFanArray。这些类为带数组引入了索引，并将带数组和索引数组的特性结合起来。可以在构造函数或者在如下的方法中，指定stripIndexCounts数组用于定义各个不同的带：

```
void setStripIndexCounts(int[] stripIndexCounts)
```

下面的代码创建了一个IndexedTriangleStripArray对象的实例：

```
int[] stripIndexCounts = {4, 4};
IndexedTriangleStripArray itsa = new IndexedTriangleStripArray(7,
    GeometryArray.COORDINATES, 8, stripIndexCounts);
Point3f[] coords = new Point3f[7];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(0f, 1f, 0f);
coords[2] = new Point3f(1f, 1f, 0f);
coords[3] = new Point3f(2f, 1f, 0f);
coords[4] = new Point3f(-1f, 0f, 0f);
coords[5] = new Point3f(-1f, -1f, 0f);
coords[6] = new Point3f(-2f, -1f, 0f);
itsa.setCoordinates(0, coords);
int[] indices = {0, 1, 2, 3, 0, 4, 5, 6};
itsa.setCoordinateIndices(0, indices);
```

其结果图形如图6-11所示。

程序清单6-1使用IndexedTriangleArray类构造了一个正四面体。正四面体是五种正多面体（也叫Platonic solids，柏拉图实体）中的一种，程序将该四面体定义为IndexedTriangleArray的一个子类，正四面体是由四个全等的等边三角形面组成的实体。程序清单6-2是一个测试程序，该程序创建了一个在空间中旋转的正四面体实例，可从不同角度

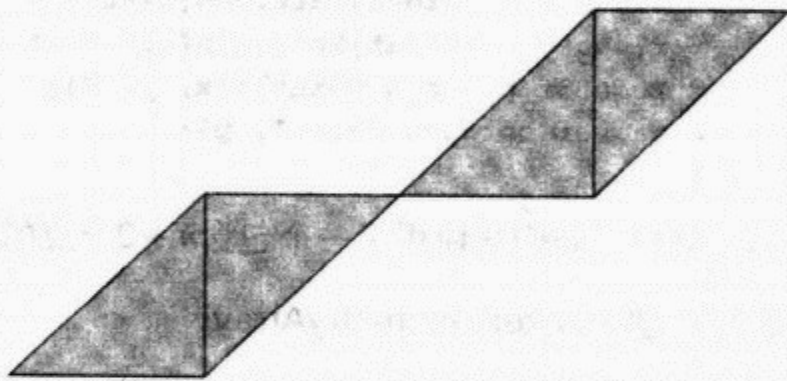


图6-11 一个由IndexedTriangleStripArray构造的图形

程序清单6-1 Tetrahedron.java

```
1 package chapter6;
2
3 import javax.vecmath.*;
4 import javax.media.j3d.*;
5 //定义Tetrahedron类，继承自IndexedTriangleArray类
6 public class Tetrahedron extends IndexedTriangleArray {
7     public Tetrahedron() {
8         super(4, TriangleArray.COORDINATES | TriangleArray.NORMALS, 12);
9         setCoordinate(0, new Point3f(1f,1f,1f)); //开始设定顶点坐标
10        setCoordinate(1, new Point3f(1f,-1,-1f));
11        setCoordinate(2, new Point3f(-1f,1f,-1f));
12        setCoordinate(3, new Point3f(-1f,-1f,1f));
13        int[] coords = {0,1,2,0,3,1,1,3,2,2,3,0}; //设置索引数组
14        float n = (float)(1.0/Math.sqrt(3));
15        setNormal(0, new Vector3f(n,n,-n)); //开始设定标准化法向量
16        setNormal(1, new Vector3f(n,-n,n));
17        setNormal(2, new Vector3f(-n,-n,-n));
```



```

18     setNormal(3, new Vector3f(-n,n,n));
19     int[] norms = {0,0,0,1,1,1,2,2,2,3,3,3};
20     setCoordinateIndices(0, coords);           //设置顶点坐标索引数组
21     setNormalIndices(0, norms);               //设置法向量索引数组
22 }
23 }

```

程序清单6-2 TestTetrahedron.java

```

1 package chapter6;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义TestTetrahedron类, 继承自Applet
12 public class TestTetrahedron extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new TestTetrahedron(), 640, 480); //创建主窗口, 设定大小
15     }
16     //重写Applet初始化函数
17     public void init() {
18         //创建Canvas3D实例
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
21         Canvas3D cv = new Canvas3D(gc);
22         setLayout(new BorderLayout()); //设置布局管理器
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph(); //创建BranchGroup
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse对象
27         su.getViewingPlatform().setNominalViewingTransform(); //设置SimpleUniverse
28         su.addBranchGraph(bg); //将BranchGroup对象加入到SimpleUniverse中
29     }
30     //生成BranchGroup的私有方法
31     private BranchGroup createSceneGraph() {
32         BranchGroup root = new BranchGroup();
33         TransformGroup spin = new TransformGroup();
34         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
35         root.addChild(spin);
36         //创建形体
37         Appearance ap = new Appearance();
38         ap.setMaterial(new Material());
39         Shape3D shape = new Shape3D(new Tetrahedron(), ap);
40         //旋转形体
41         Transform3D tr = new Transform3D();
42         tr.setScale(0.25);
43         TransformGroup tg = new TransformGroup(tr);
44         spin.addChild(tg);
45         tg.addChild(shape);

```



```

46 Alpha alpha = new Alpha(-1, 4000);
47 RotationInterpolator rotator =
48 new RotationInterpolator(alpha, spin);
49 BoundingSphere bounds = new BoundingSphere();
50 rotator.setSchedulingBounds(bounds);
51 spin.addChild(rotator);
52 //设置光照和背景
53 Background background = new Background(1.0f, 1.0f, 1.0f);
54 background.setApplicationBounds(bounds);
55 root.addChild(background);
56 AmbientLight light = new AmbientLight
57 (true, new Color3f(Color.red));
58 light.setInfluencingBounds(bounds);
59 root.addChild(light);
60 PointLight ptlight = new PointLight(new Color3f(Color.green),
61 new Point3f(3f,3f,3f), new Point3f(1f,0f,0f));
62 ptlight.setInfluencingBounds(bounds);
63 root.addChild(ptlight);
64 PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
65 new Point3f(-2f,2f,2f), new Point3f(1f,0f,0f));
66 ptlight2.setInfluencingBounds(bounds);
67 root.addChild(ptlight2);
68 return root;
69 }
70 }

```



图6-12 一个正四面体实例

在上面的例子中，定义了两个类：Tetrahedron和TestTetrahedron。Tetrahedron继承自IndexedTriangleArray类（代码第6行），因此它也可直接用做场景图中某个Shape3D节点的几何属性节点。这个正四面体的顶点坐标如下：

(1, 1, 1), (1, -1, -1), (-1, 1, -1), (-1, -1, 1)

四个三角形面由12个指向顶点坐标数组的索引确定：

0, 1, 2, 0, 3, 1, 1, 3, 2, 2, 3, 0

四个面的法向量由如下四个向量确定：

(1, 1, -1), (1, -1, 1), (-1, -1, -1), (-1, 1, 1)

这些向量除以 $\sqrt{3}$ 后,可得到标准向量。每一个面的每一个顶点都是确定的,类似于每一条法向量都对应于相关面。

TestTetrahedron类是一个典型的Java 3D applet/application程序,用于测试Tetrahedron类,它创建了Tetrahedron的一个实例,并将其与一个Shape3D节点相关联。图6-13为该程序对应的场景图。

场景图中的该四面体由一个Shape3D对象表示(代码第39行)。它引用了一个Tetrahedron对象实例作为其几何属性,还引用了一个Appearance对象,Appearance对象则引用了一个Material对象以使用光照。

该Shape3D节点附属到一个TransformGroup节点,以支持对四面体的缩放操作。该TransformGroup节点附属到另一个TransformGroup节点,而这个TransformGroup节点由一个称为RotationInterpolator的Behavior对象控制,执行旋转变换。

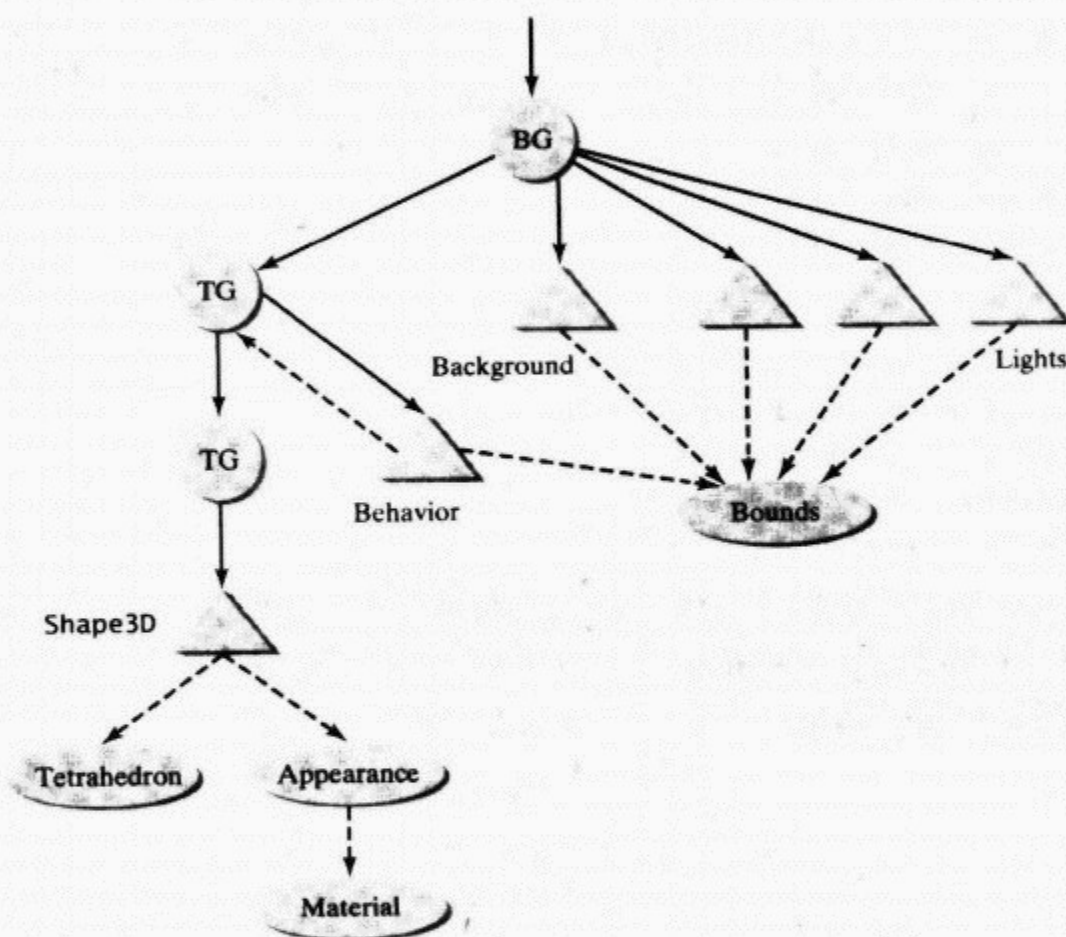


图6-13 程序清单6-2对应的场景图

182

场景图定义了一个背景节点,创建了一个白色的背景,定义了三个不同的光源节点对场景进行光照。此外,行为(behavior)、背景(background)和光照(light)节点都共享同一个Bounds对象(一个半径为1.0的球体)。

6.3.4 法向量

在某些复杂的绘制模式(如光照)下,曲面的法向量是重要的几何属性。平面的法向量(normal)是一个垂直于平面的向量(如图6-14所示)。曲面在某一点的法向量,是曲面在这一点的切平面的法向量。

两个向量的外积也称为叉积(cross product)与这两个向量都垂直,因此向量的外积经常用于计算法向量。例如,给定平面中的三个点 P_0 、 P_1 和 P_2 ,我们可以得到平面中的两个向量 $v_1 = P_1 - P_0$ 以及 $v_2 = P_2 - P_0$ (如图6-15)。两个向量的外积 $v_1 \times v_2$ 即为平面的法向量。

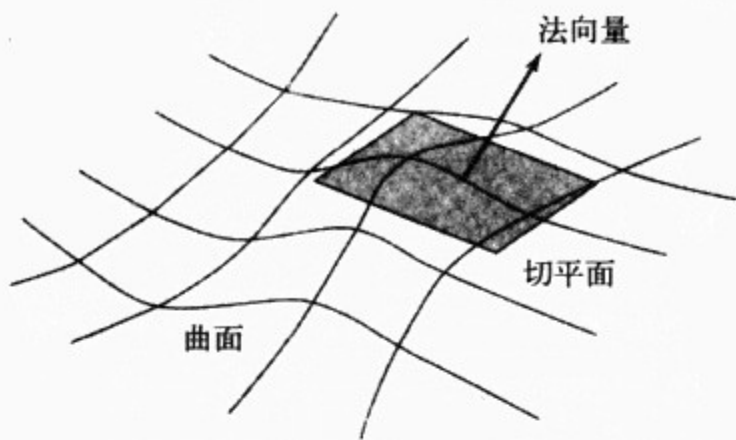


图6-14 曲面的法向量垂直于切平面

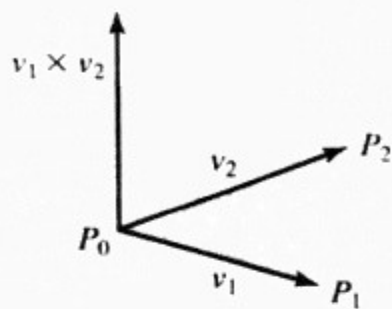


图6-15 使用外积计算法向量

曲面的法向量通常表示为单位向量（长度为1的向量）。任意非零向量 v 都可以使用公式 $v/||v||$ 转换为单位向量，这称为向量的归一化（normalization）。

Vector3f和Vector3D类都有cross方法用于计算两个3D向量间的外积，normalize方法用于向量归一化。例如，下面的代码计算了由三个点确定的平面的法向量：

```
Point3f p0 = new Point3f(1, 1, 1);
Point3f p1 = new Point3f(1, -1, -1);
Point3f p2 = new Point3f(-1, 1, -1);
p1.sub(p0);
p2.sub(p0);
Vector3f v1 = new Vector3f(p1);
Vector3f v2 = new Vector3f(p2);
Vector3f normal = new Vector3f();
normal.cross(v1, v2); //计算向量外积
normal.normalize(); //向量归一化
```

183

对于一个由多个平面构成的几何体，例如四面体，每一个面的法向量可分别通过上述方法计算得出。对于一个用多边形网格拟合的光滑曲面而言，在顶点处的法向量应该从原始曲面而不是根据多边形进行计算。考虑一个由如下参数方程表示的曲面：

$$x = f(u, v)$$

$$y = g(u, v)$$

$$z = h(u, v)$$

求导，可以得出切平面上的两个向量：

$$(dx/du, dy/du, dz/du) = (f_u, g_u, h_u)$$

$$(dx/dv, dy/dv, dz/dv) = (f_v, g_v, h_v)$$

曲面在某一点的法向量，可通过计算它们的外积得到：

$$n = (f_u, g_u, h_u) \times (f_v, g_v, h_v)$$

例如，一个椭圆抛物面有如下的参数方程：

$$x = u \cos v$$

$$y = u \sin v$$

$$z = u^2$$

其偏导如下：

$$(\cos v, \sin v, 2u)$$

$$(-u \sin v, u \cos v, 0)$$

因此曲面在点 (u, v) 处的法向量为:

$$(\cos v, \sin v, 2u) \times (-u \sin v, u \cos v, 0) = (-2u^2 \cos v, 2u^2 \sin v, u)$$

曲面和法向量的相关数学背景知识, 参见附录A。

6.4 类GeometryInfo

6.4.1 使用GeometryInfo类

除了GeometryArray类族, 我们还可以使用GeometryInfo类来构造几何体, 这个类并没有部分设置顶点数据的方法, 但可用于定义更一般的多边形面。通过使用工具类NormalGenerator和Stripifier, 还能自动生成曲面的法向量, 并条纹化(stripify) GeometryInfo对象。

GeometryInfo类有一个构造函数以GeometryArray对象为参数, 这提供了从GeometryArray对象到GeometryInfo对象的转化:

```
public GeometryInfo(GeometryArray ga)
```

GeometryInfo类的另一个构造函数, 使用primitiveType参数构造空对象:

```
public GeometryInfo(int primitiveType)
```

其中primitiveType的值可为:

```
TRIANGLE_ARRAY  
TRIANGLE_FAN_ARRAY  
TRIANGLE_STRIP_ARRAY  
QUAD_ARRAY  
POLYGON_ARRAY
```

三角形和四边形数组的解释与相应的GeometryArray类相同, 设置顶点数据时, 必须提供整个数组数据, 例如:

```
void setCoordinates(Point3f[] coords)
```

也可以很简单地设置一个索引数值, 实现索引表示:

```
void setCoordinateIndices(int[] indices)  
void setNormalIndices(int[] indices)  
void setColorIndices(int[] indices)  
void setTextureCoordinateIndices(int[] indices)
```

与GeometryArray类似, stripCounts数组用于为设置了TRIANGLE_FAN_ARRAY和TRIANGLE_STRIP_ARRAY标志位的GeometryInfo对象定义独立的带。

带POLYGON_ARRAY标志位的多边形数组, 可以用来定义超过四条边且带有空洞的复杂多边形。每个这样的多边形由一个或多个轮廓组成, 第一个轮廓定义了多边形的外边界, 其他的轮廓定义了多边形内部的空洞。

带POLYGON_ARRAY标志位的多边形数组的stripCounts数组, 定义了每个轮廓的顶点数目。contourCounts指定了每一个多边形的轮廓数目, 例如, 图6-16显示了给定多边形数组的带数和轮廓的数目。

GeometryInfo类会在内部将一个POLYGON_ARRAY几何基元中的多边形转换为一系列三角形, 这个三角化的过程是通过Triangulator类自动进行的。

GeometryInfo对象的曲面法向量, 可以使用NormalGenerator类自动计算得到。如果要考虑光源的光照效果, 曲面法向量的计算对绘制过程中是必需的。法向量的自动生成, 有助于减少繁琐的计算。NormalGenerator类中, 有一个计算GeometryInfo对象法向量的方法:

184

185


```
Public void generateNormals(GeometryInfo gi);
```

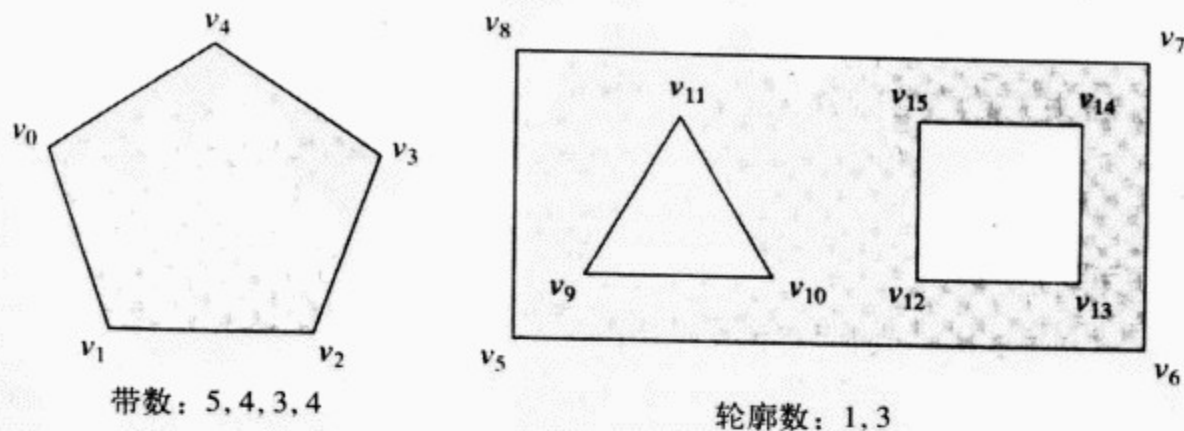


图6-16 使用GeometryInfo类构造几何体

为了得到GeometryInfo对象gi的法向量，需要创建一个NormalGenerator实例，并调用其generateNormals方法：

```
NormalGenerator ng = new NormalGenerator();
ng.generateNormals(gi);
```

另一个辅助类Stripifier，有助于将一个GeometryInfo对象的几何属性转变为三角形带。Stripifier的使用与NormalGenerator类似：

```
Stripifier st = new Stripifier();
st.stripify(gi);
```

程序清单6-3通过使用GeometryInfo类，定义了一个正十二面体（另一种柏拉图立体），该程序演示了GeometryInfo类的使用方法。程序清单6-4为相应的测试程序。

程序清单6-3 Dodecahedron.java

```
1 package chapter6;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.geometry.*;
8 //定义Dodecahedron类，继承自Shape3D类
9 public class Dodecahedron extends Shape3D{
10     public Dodecahedron() {
11         GeometryInfo gi = new GeometryInfo(GeometryInfo.POLYGON_ARRAY);
12         double phi = 0.5*(Math.sqrt(5)+1);
13         Point3d[] vertices = {new Point3d(1,1,1),
14                                new Point3d(0,1/phi,phi),
15                                new Point3d(phi,0,1/phi),new Point3d(1/phi,phi,0),
16                                new Point3d(-1,1,1),
17                                new Point3d(0,-1/phi,phi),new Point3d(1,-1,1),
18                                new Point3d(phi,0,-1/phi),
19                                new Point3d(1,1,-1),new Point3d(-1/phi,phi,0),
20                                new Point3d(-phi,0,1/phi),
21                                new Point3d(-1,-1,1),new Point3d(1/phi,-phi,0),
22                                new Point3d(1,-1,-1),
23                                new Point3d(0,1/phi,-phi),new Point3d(-1,1,-1),
24                                new Point3d(-1/phi,-phi,0),
25                                new Point3d(-phi,0,-1/phi),new Point3d(0,-1/phi,-phi),
```



```

26     new Point3d(-1,-1,-1));
27     int[] indices = {0,1,5,6,2, 0,2,7,8,3, 0,3,9,4,1,      //索引数组
28     1,4,10,11,5, 2,6,12,13,7, 3,8,14,15,9,
29     5,11,16,12,6, 7,13,18,14,8, 9,15,17,10,4,
30     19,16,11,10,17, 19,17,15,14,18, 19,18,13,12,16};
31     gi.setCoordinates(vertices); //设置gi顶点坐标数组
32     gi.setCoordinateIndices(indices); //设置gi顶点坐标索引数组
33     int[] stripCounts = {5,5,5,5,5,5,5,5,5,5,5,5};
34     gi.setStripCounts(stripCounts);          //设置gi带数目数组
35     NormalGenerator ng = new NormalGenerator();
36     ng.generateNormals(gi);                  //自动生成法向量
37     this.setGeometry(gi.getGeometryArray());
38 }
39 }

```

程序清单6-4 TestDodecahedron.java

```

1 package chapter6;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义TestDodecahedron类,继承自Applet类
12 public class TestDodecahedron extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new TestDodecahedron(), 640, 480); //创建主窗口,设定大小
15     }
16     //重写Applet初始化函数
17     public void init() {
18         GraphicsConfiguration gc =
19             SimpleUniverse.getPreferredConfiguration();
20         //创建GraphicsConfiguration对象
21         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D实例
22         setLayout(new BorderLayout()); //设置布局管理器
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph(); //创建BranchGroup对象
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse对象
27         su.getViewingPlatform().setNominalViewingTransform(); //设置SimpleUniverse
28         su.addBranchGraph(bg); //将BranchGroup对象加入到SimpleUniverse中
29     }
30     //生成BranchGroup的私有方法
31     private BranchGroup createSceneGraph() {
32         BranchGroup root = new BranchGroup();
33         TransformGroup spin = new TransformGroup();
34         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
35         root.addChild(spin);
36         //创建形体

```



```

37  Appearance ap = new Appearance();
38  ap.setMaterial(new Material());
39  Shape3D shape = new Dodecahedron();
40  shape.setAppearance(ap);
41  //形体变换
42  Transform3D tr = new Transform3D();
43  tr.setScale(0.25);
44  TransformGroup tg = new TransformGroup(tr);
45  spin.addChild(tg);
46  tg.addChild(shape);
47
48  Alpha alpha = new Alpha(-1, 4000);
49  RotationInterpolator rotator =
50  new RotationInterpolator(alpha, spin);
51  BoundingSphere bounds = new BoundingSphere();
52  rotator.setSchedulingBounds(bounds);
53  spin.addChild(rotator);
54
55  //设置背景和光照
56  Background background = new Background(1.0f, 1.0f, 1.0f);
57  background.setApplicationBounds(bounds);
58  root.addChild(background);
59  AmbientLight light = new AmbientLight
60      (true, new Color3f(Color.red));
61  light.setInfluencingBounds(bounds);
62  root.addChild(light);
63  PointLight ptlight = new PointLight(new Color3f(Color.green),
64      new Point3f(3f,3f,3f), new Point3f(1f,0f,0f));
65  ptlight.setInfluencingBounds(bounds);
66  root.addChild(ptlight);
67  PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
68      new Point3f(-2f,2f,2f), new Point3f(1f,0f,0f));
69  ptlight2.setInfluencingBounds(bounds);
70  root.addChild(ptlight2);
71  return root;
72  }
73  }

```

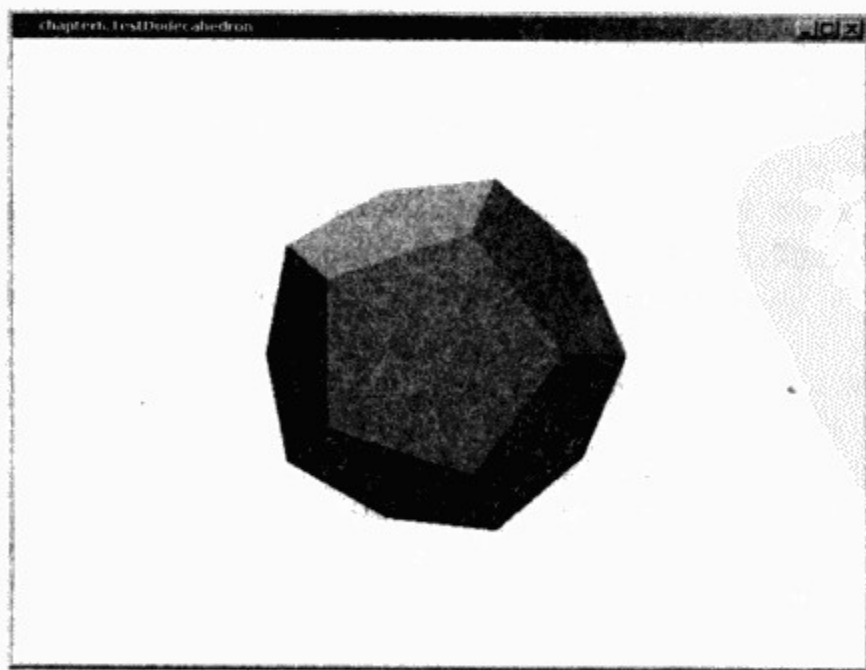


图6-17 一个正十二面体

程序中, Dodecahedron类定义为Shape3D类的子类, 该十二面体有20个顶点及12个五边形面。程序中创建了一个GeometryInfo对象, 以定义这些面, 并自动计算出法向量。顶点坐标由数组vertices直接设定, 20个顶点的坐标值为:

```
(1, 1, 1),
(0, 1/φ, φ), (φ, 0, 1/φ), (1/φ, φ, 0),
(-1, 1, 1), (0, -1/φ, φ), (1, -1, 1),
(φ, 0, -1/φ), (1, 1, -1), (-1/φ, φ, 0),
(-φ, 0, 1/φ), (-1, -1, 1), (1/φ, -φ, 0),
(1, -1, -1), (0, 1/φ, -φ), (-1, 1, -1),
(-1/φ, -φ, 0), (-φ, 0, -1/φ), (0, -1/φ, -φ),
(-1, -1, -1)
```

188

其中 $\phi = (\sqrt{5} + 1)/2$ 。十二面体的12个面分别由如下指向顶点的索引元组定义:

```
(0, 1, 5, 6, 2), (0, 2, 7, 8, 3), (0, 3, 9, 4, 1),
(1, 4, 10, 11, 5), (2, 6, 12, 13, 7), (3, 8, 14, 15, 9),
(5, 11, 16, 12, 6), (7, 13, 18, 14, 8), (9, 15, 17, 10, 4),
(19, 16, 11, 10, 17), (19, 17, 15, 14, 18), (19, 18, 13, 12, 16)
```

数组stripCount (第33行) 定义了多边形的数目, 以及每一个多边形面的顶点数目:

```
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5
```

程序中创建了一个NormalGenerator对象, 用于计算GeometryInfo对象各面的法向量 (第35到36行)。最终由GeometryInfo对象所构成的几何体, 用于定义当前Shape3D对象的几何属性。

测试程序TestDodecahedron与程序清单6-2几乎相同, 唯一主要区别在于, Dodecahedron对象是直接作为叶节点添加到场景图中, 因为其本身是一个Shape3D对象。

6.4.2 创建多边形网格

一个普通的曲面往往需要大量的小多边形片, 才能获得比较好的拟合。下面的参数方程所定义的曲面:

$$\begin{aligned}x &= f(u, v) \\y &= g(u, v) \\z &= h(u, v)\end{aligned}$$

其中两个参数的定义域为矩形 $a \leq u \leq b$, $c \leq v \leq d$ 。通常, 可以将参数范围分成 $m \times n$ 的网格来生成分片 (patche)。网格的顶点定义如下:

$$\begin{aligned}u_i &= a + i(b-a)/m, i = 0, 1, 2, \dots, m \\v_j &= c + j(d-c)/n, j = 0, 1, 2, \dots, n\end{aligned}$$

一个四边形片 (quadrilateral patch) 可以由对应以下参数值的四个顶点定义:

$$(u_i, v_j), (u_{i+1}, v_j), (u_{i+1}, v_{j+1}), (u_i, v_{j+1})$$

四边形片还可以再细分为两个三角形:

$$(u_i, v_j), (u_{i+1}, v_j), (u_{i+1}, v_{j+1}) \text{ 和 } (u_i, v_j), (u_{i+1}, v_{j+1}), (u_i, v_{j+1})$$

由于相邻多边形间顶点的重叠现象比较明显, 通常会使用索引数组来提高效率。

GeometryInfo和GeometryArray类中设置顶点坐标的方法不接受2D数组参数, 因此网格中的顶点必须定义为线性格式。

程序清单6-5演示了一个曲面的多边形网格拟合的生成过程,该程序使用了GeometryInfo类从一个二元函数的采样点集生成几何体。

数据可视化是计算机图形学的一个重要应用,随着数据集维数的增大,对数据的分析和理解也变得更难,数据的3D图形绘制能极大地改善数据集的表现形式。该程序实例演示把如下二元函数绘制为3D曲面:

$$y = 2\cos(x^2)\sin(z^2) / e^{0.25(x^2+z^2)} - 1$$

尽管实例中使用的数据集产生于一个给定的函数,但是该方法可以很方便地应用到其他数据源。图6-18为生成的结果,绘图结果一直围绕着纵轴旋转。

程序清单6-5 ViewData.java

```

1 package chapter6;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义ViewData类,继承自Applet类
12 public class ViewData extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new ViewData(), 640, 480); //创建主窗口,设定窗口大小
15     }
16     //重写Applet初始化函数
17     public void init() {
18         //创建canvas
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
21         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D对象
22         setLayout(new BorderLayout()); //设置布局管理器
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph(); //创建BranchGroup对象
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv); //生成SimpleUniverse对象
27         su.getViewingPlatform().setNominalViewingTransform(); //设置SimpleUniverse
28         su.addBranchGraph(bg); //将BranchGroup对象加入SimpleUniverse
29     }
30     //生成BranchGroup对象的私有方法
31     private BranchGroup createSceneGraph() {
32         BranchGroup root = new BranchGroup();
33         TransformGroup spin = new TransformGroup();
34         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
35         root.addChild(spin);
36
37         //生成形体
38         Appearance ap = new Appearance();
39         ap.setMaterial(new Material());
40         Shape3D shape = new Shape3D(createGeometry(), ap);
41         //形体变换

```



```

42 Transform3D tr = new Transform3D();
43 tr.setScale(0.2);
44 TransformGroup tg = new TransformGroup(tr);
45 spin.addChild(tg);
46 tg.addChild(shape);
47
48 Alpha alpha = new Alpha(-1, 12000);
49 RotationInterpolator rotator = new RotationInterpolator
50     (alpha, spin);
51 BoundingSphere bounds = new BoundingSphere();
52 rotator.setSchedulingBounds(bounds);
53 spin.addChild(rotator);
54
55 //设置背景和光照
56 Background background = new Background(1.0f, 1.0f, 1.0f);
57 background.setApplicationBounds(bounds);
58 root.addChild(background);
59 AmbientLight light = new AmbientLight
60     (true, new Color3f(Color.red));
61 light.setInfluencingBounds(bounds);
62 root.addChild(light);
63 PointLight ptlight = new PointLight(new Color3f(Color.green),
64     new Point3f(3f,3f,3f), new Point3f(1f,0f,0f));
65 ptlight.setInfluencingBounds(bounds);
66 root.addChild(ptlight);
67 return root;
68 }
69 //生成Geometry的私有方法
70 private Geometry createGeometry() {
71     int m = 40;
72     int n = 40;
73     Point3f[] pts = new Point3f[m*n];
74     int idx = 0;
75     for (int i = 0; i < m; i++) {
76         for (int j = 0; j < n; j++) {
77             float x = (i - m/2)*0.2f;
78             float z = (j - n/2)*0.2f;
79             float y = 2f * (float)(Math.cos(x*x) * Math.sin(z*z))/
80                 ((float)Math.exp(0.25*(x*x+z*z)))-1.0f;
81             pts[idx++] = new Point3f(x, y, z);
82         }
83     }
84
85     int[] coords = new int[2*n*(m-1)];
86     idx = 0;
87     for (int i = 1; i < m; i++) {
88         for (int j = 0; j < n; j++) {
89             coords[idx++] = i*n + j;
90             coords[idx++] = (i-1)*n + j;
91         }
92     }
93
94     int[] stripCounts = new int[m-1];
95     for (int i = 0; i < m-1; i++) stripCounts[i] = 2*n;

```



```

96
97     GeometryInfo gi = new GeometryInfo
98         (GeometryInfo.TRIANGLE_STRIP_ARRAY); //构造GeometryInfo实例
99     gi.setCoordinates(pts);
100    gi.setCoordinateIndices(coords);
101    gi.setStripCounts(stripCounts);
102
103    NormalGenerator ng = new NormalGenerator();
104    ng.generateNormals(gi); //生成法向量
105
106    return gi.getGeometryArray();
107 }
108 }

```

191

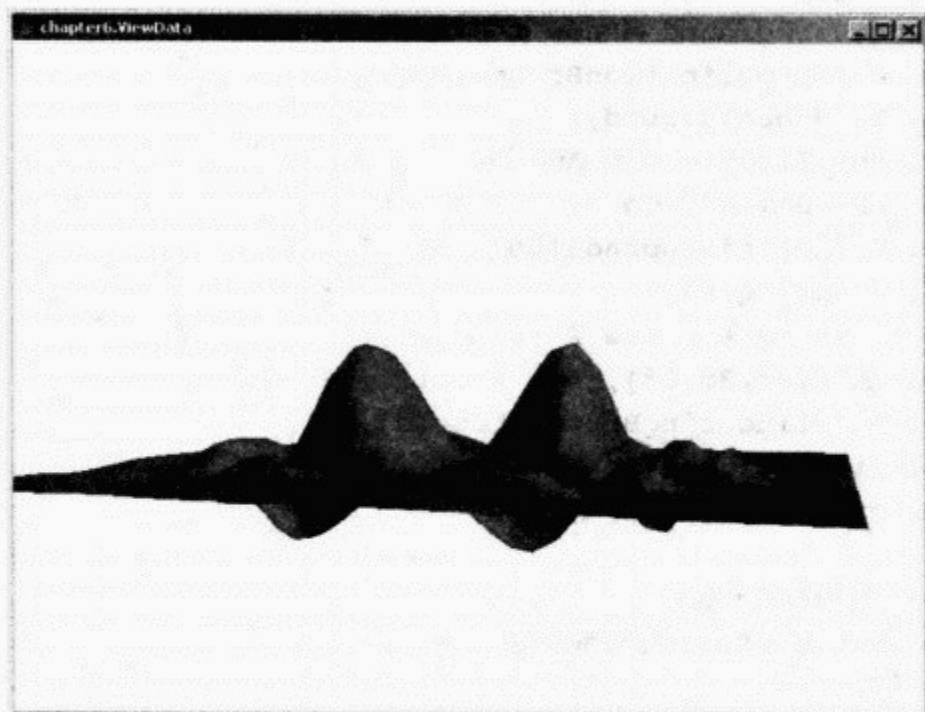


图6-18 一个二元函数的3D数据图

我们使用从上述函数生成的采样点来构造几何体，在 xz 平面上创建了一个 $m \times n$ 的矩形网格来对函数求值。函数值对应于点的 y 坐标值，这些点保存在一维数组`pts`中（第73行）。这个网格点形成的曲面定义为一组三角形带，每两个相邻行的点构成了一个三角形带。

该图形表面的几何属性由`GeometryInfo`对象（几何基元类型为`TRIANGLE_STRIP_ARRAY`）定义（第98行）。`GeometryInfo`对象是用于进行索引的，数组`pts`定义了顶点坐标，实际的三角形带组由一个单独的索引数组`coords`和一个带数数组`stripCounts`来定义。数组`coords`通过指向顶点数组的索引定义了各三角形带，因为网格在 x 轴方向上有 m 个值，因此会产生 $m-1$ 个带，每一个带由两行的 $2n$ 个点定义得到。因此，数组`coords`中将存放 $2n(m-1)$ 个索引，相邻两行的索引交替存放。例如，第一个带的索引值为：

$$n, 0, n+1, n+2, 2, \dots$$

第二个带的索引值为：

$$2n, n, 2n+1, 2n+2, n+2, \dots$$

所有的索引值都存放在同一个数组`coords`中，为了将一维的索引数据分成多个带，我们使用数组`stripCounts`来确定共有 $m-1$ 个带，每个带中有 $2n$ 个索引值。

程序中使用了`NormalGenerator`工具类来生成曲面的法向量，所得到的几何体被一个`Shape3D`对象作为几何属性使用，用于定义曲面。

该实例的场景图与程序6-2的场景图相似，程序中只用到了两个光源。

192

6.5 几何基元

Java 3D中，针对常用的几何基元封装了一些工具类，以方便用户进行编程。抽象类Primitive是Group的子类，封装了预定义的一些几何属性。这些类使用独立的低层次结构如GeometryArray或GeometryInfo对象，而被用做场景图中的高层组件，不需为其设置复杂的几何属性。

图6-19显示了com.sun.j3d.utils.geometry包中几何基元类的类层次结构。几何基元的外观可通过如下方法进行设置：

```
void setAppearance()  
void setAppearance(Appearance appearance)  
void setAppearance(int subpart, Appearance appearance)
```

几何基元的大小可通过它们的构造函数进行设置，例如，

```
Box(float xdim, float ydim, float zdim, Appearance appearance)  
Cone(float radius, float height)  
Cylinder(float radius, float height)  
Sphere(float radius)
```

因为Primitive类是Group的子类，一个几何基元对象可直接作为一个节点加入到场景图中。

程序清单6-6演示了几何基元的应用，该程序展示了由Java 3D工具包所提供的四个几何基元类的实例（见图6-20）。

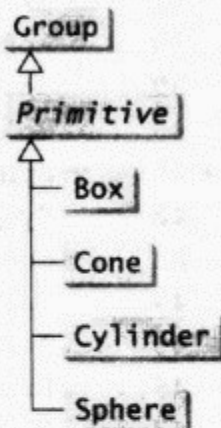


图6-19 几何基元类

程序清单6-6 TestPrimitives.java

```
1 package chapter6;  
2  
3 import javax.vecmath.*;  
4 import java.awt.*;  
5 import java.awt.event.*;  
6 import javax.media.j3d.*;  
7 import com.sun.j3d.utils.universe.*;  
8 import com.sun.j3d.utils.geometry.*;  
9 import java.applet.*;  
10 import com.sun.j3d.utils.applet.MainFrame;  
11 //定义TestPrimitives类，继承自Applet  
12 public class TestPrimitives extends Applet {  
13     public static void main(String[] args) {  
14         new MainFrame(new TestPrimitives(), 640, 480); //生成主窗口，设置窗口大小  
15     }  
16     //重写Applet初始化函数  
17     public void init() {  
18         //创建Canvas3D对象  
19         GraphicsConfiguration gc =  
20             SimpleUniverse.getPreferredConfiguration();  
21         Canvas3D cv = new Canvas3D(gc);  
22         setLayout(new BorderLayout()); //设置布局  
23         add(cv, BorderLayout.CENTER);  
24         BranchGroup bg = createSceneGraph(); //创建BranchGroup对象  
25         bg.compile();
```

193


```

26     SimpleUniverse su = new SimpleUniverse(cv); //创建设置SimpleUniverse对象
27     su.getViewingPlatform().setNominalViewingTransform();
28     su.addBranchGraph(bg);
29 }
30 //生成BranchGroup的私有方法
31 private BranchGroup createSceneGraph() {
32     BranchGroup root = new BranchGroup();
33     TransformGroup spin = new TransformGroup();
34     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
35     root.addChild(spin);
36
37     //开始创建几何基元对象
38     Appearance ap = new Appearance();
39     ap.setMaterial(new Material());
40     Box box = new Box(1.2f, 0.3f, 0.8f, ap); //长方体
41     Sphere sphere = new Sphere(); //球体
42     Cylinder cylinder = new Cylinder(); //圆柱体
43     Cone cone = new Cone(); //圆锥体
44     //创建并设置四个形体的TransformGroup节点对象
45     Transform3D tr = new Transform3D();
46     tr.setScale(0.2);
47     TransformGroup tg = new TransformGroup(tr);
48     spin.addChild(tg);
49     tg.addChild(box);
50     tr.setIdentity();
51     tr.setTranslation(new Vector3f(0f, 1.5f, 0f));
52     TransformGroup tgSphere = new TransformGroup(tr);
53     tg.addChild(tgSphere);
54     tgSphere.addChild(sphere);
55     tr.setTranslation(new Vector3f(-1f, -1.5f, 0f));
56     TransformGroup tgCylinder = new TransformGroup(tr);
57     tg.addChild(tgCylinder);
58     tgCylinder.addChild(cylinder);
59     tr.setTranslation(new Vector3f(1f, -1.5f, 0f));
60     TransformGroup tgCone = new TransformGroup(tr);
61     tg.addChild(tgCone);
62     tgCone.addChild(cone);
63     //设置旋转
64     Alpha alpha = new Alpha(-1, 4000);
65     RotationInterpolator rotator =
66     new RotationInterpolator(alpha, spin);
67     BoundingSphere bounds = new BoundingSphere();
68     rotator.setSchedulingBounds(bounds);
69     spin.addChild(rotator);
70
71     //设置背景和光照
72     Background background = new Background(1.0f, 1.0f, 1.0f);
73     background.setApplicationBounds(bounds);
74     root.addChild(background);
75     AmbientLight light =
76     new AmbientLight(true, new Color3f(Color.red)); //设置环境光源
77     light.setInfluencingBounds(bounds);
78     root.addChild(light);
79     PointLight ptlight = new PointLight(new Color3f(Color.green),
80     new Point3f(3f, 3f, 3f), new Point3f(1f, 0f, 0f)); //创建点光源

```



```

81 ptlight.setInfluencingBounds(bounds);
82 root.addChild(ptlight);
83 PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
84   new Point3f(-2f,2f,2f), new Point3f(1f,0f,0f)); //创建点光源
85 ptlight2.setInfluencingBounds(bounds);
86 root.addChild(ptlight2);
87 return root;
88 }
89 }

```



图6-20 四个几何基元

该程序创建了四种几何基元（长方体、球体、圆柱体和圆锥体）的实例，并将它们放置到场景图中，其场景图如图6-21所示。三个几何基元点上的几何变换节点，用于在虚拟世界中把这些实体相互分开。第7章将详细介绍如何应用几何变换节点。

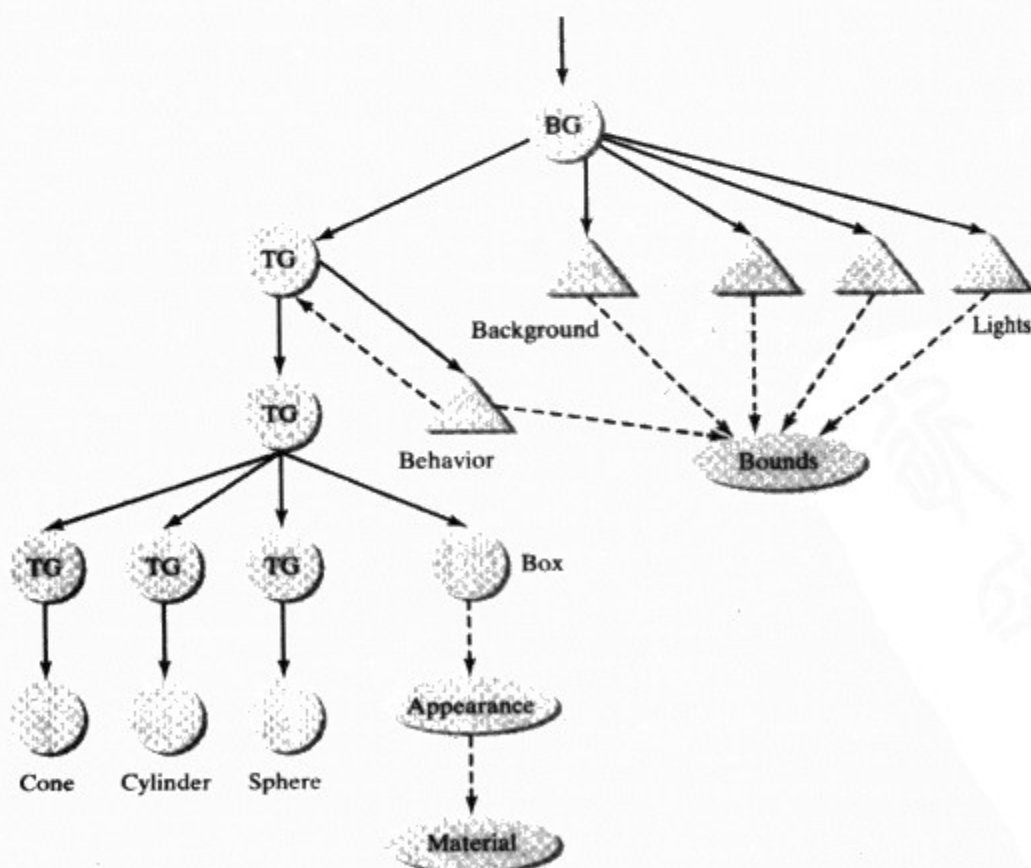


图6-21 程序清单6-4的场景图

程序使用了默认构造函数创建Cone、Sphere及Cylinder对象，为光照效果创建并设置了这些对象的表面法向量。Box对象由一个带参数的构造函数创建，构造时指定了它的x、y和z维度的大小及外观（第40到43行）。

这四个对象都附属到一个TransformGroup节点，该节点执行缩放操作。整个场景由一个RotationInterpolator对象通过一个TransformGroup节点实现旋转操作，背景和光照的配置与前面的例子相似。

6.6 字体和文本

字体为构造几何体提供了丰富的资源，文本字符串的字形是定义在字体中的复杂的几何体。

Java 3D中包含了Text3D和Text2D类。Text3D类继承自Geometry类，因此一个Text3D对象也可以被Shape3D节点引用作为几何属性，一个Text3D对象由3D字体以及文本的字符串确定。Font3D对象是3D版本的字体，Font3D对象是从普通2D AWT字体和由FontExtrusion类定义的凸出量一起构造的。如下是一个典型的Text3D对象创建过程：

- 创建一个java.awt.Font对象。
- 创建一个FontExtrusion对象。
- 使用Font和FontExtrusion对象创建Font3D对象。
- 使用Font3D对象和一个String对象创建Text3D对象。

例如，下面的语句创建了一个Text3D对象，其文本为“Hello”，字体为粗体Serif，字体大小为1，使用默认的凸出量。

```
Font font = new Font("Serif", Font.BOLD, 1);
FontExtrusion extrusion = new FontExtrusion();
Font3D font3d = new Font3D(font, extrusion);
Text3D text = new Text3D(font3d, "Hello");
```

Text2D是Shape3D的子类，因此可直接作为场景图的一个叶节点。Text2D对象作为一个矩形加以实现，而文本字符串作为一幅图像绘制在矩形上。下列代码定义了一个字符串为“Hello”的Text2D对象，字体为斜体Serif，大小为16，颜色为蓝色：

```
Text2D text = new Text2D("Hello", Color.blue, "Serif", 16, Font.Italic);
```

6.7 外观和属性

除了几何属性，几何对象还具有定义其绘制时的外观的属性。几何对象的外观有多种不同模型，它们在绘制质量和效率上有很大的不同。

颜色可能是其中一个最基本的属性。不过，有多种不同的方法为几何体各部分定义颜色，一个简单的方法就是将所有几何对象单元（如点、线或多边形面）都设置为使用单一指定的颜色。

另一种方法是采用插值的方法为不同的点设置不同的颜色，也称做Gouraud明暗处理（Gouraud shading）方法。该方法指定了多边形的每个顶点的颜色，其余点的颜色则根据顶点颜色进行插值计算得到。

更具有真实感的着色方案，既要考虑光照等环境因素的影响，也要考虑对象本身的属性。几何对象的颜色通常会受如下一些因素影响：物体的反射特性、几何体、光源、自发射光、环境光以及视点位置等。

纹理映射是一种实现复杂外观的强大技术。对于具有大量细节的对象，将外观保存为光栅图像并在绘制时把图像映射到对象表面，是一种效率更高的途径。

在Java 3D中, Appearance节点组件指定了与节点绘制相关的属性, Appearance对象引用一组属性对象。图6-22列出了与外观属性相关的类。

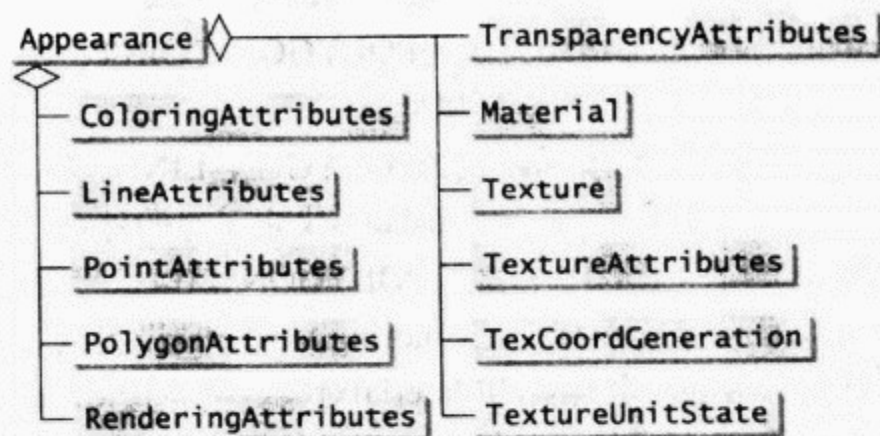


图6-22 Appearance相关类

当既没有光照也没有定义顶点颜色 (vertex color) 时, ColoringAttributes对象可用于定义与其相关联的几何对象的颜色。如果定义了几何体的顶点的颜色, 上述颜色将被忽略。如果启用了光照, 上述颜色也将被忽略。ColoringAttributes也定义了明暗处理模型 (shading model)。

- SHAD_FLAT: 平面明暗处理方案为多边形或线的每一个点, 都采用固定的颜色。
- SHAD_GOURAUD: Gouraud明暗处理方案下, 多边形或线内部的点根据顶点颜色的插值计算得到。多边形上点的颜色不是常量, 而是会平稳地变化, 以产生更逼真的外观。
- SHAD_FASTEST: 该常量选择当前平台下最快的明暗处理模型。
- SHAD_NICEST: 该常量选择当前平台下效果最佳的明暗处理模型。

PointAttributes对象定义了与点绘制相关的属性, 可以指定点的大小及反走样属性。

LineAttributes对象定义了与线绘制相关的属性, 线的属性包括线宽、线型及反走样属性, 线型由类中的以下类常量定义:

```
PATTERN_SOLID
PATTERN_DASH
PATTERN_DOT
PATTERN_DASH_DOT
PATTERN_USER_DEFINED
```

PolygonAttributes对象定义了与多边形绘制相关的属性, 多边形的绘制模式决定了其绘制方式。三种多边形绘制模式定义如下:

```
POLYGON_POINT
POLYGON_LINE
POLYGON_FILL
```

多边形绘制模式可以在构造函数中进行设置, 或者通过setPolygonMode方法进行设置。

TransparencyAttributes节点分量提供了绘制具有一定透明度 (transparency) 的可视对象的方法。例如, 下列TransparencyAttributes对象可以设定可视对象的绘制透明度值为0.6:

```
int tMode = TransparencyAttributes.BLENDED;
float tValue = 0.6f;
TransparencyAttributes ta = new TransparencyAttributes(tMode, tValue);
Appearance ap = new Appearance();
ap.setTransparencyAttributes(ta);
```

透明度值为1.0表示完全透明, 为0.0则表示完全不透明。值得注意的是, 这里的透明度值与颜色描述中的透明度 (alpha) 值恰恰相反。

外观的材质设置与对象的光照和明暗相关。第9章将讨论光照的细节，纹理映射则在第9章和第12章进行讨论。

外观的各种属性的设置是相互关联的，几何绘制模式（drawing modes）的应用规则如下：

1. 如果PolygonAttributes的多边形绘制模式为POLYGON_POINT，则只绘制多边形的顶点。在这种模式下，PointAttributes决定了点的绘制属性。
2. 如果PolygonAttributes的多边形绘制模式为POLYGON_LINE，则绘制多边形的边，可视对象将以线框形式展现。在这种模式下，LineAttributes决定了边的外观。
3. 如果PolygonAttributes的多边形绘制模式为POLYGON_FILL，则填充多边形。

198

对象的着色（coloring）依赖于光照模型、ColoringAttributes设置以及几何体的顶点数据。

1. 如果Appearance节点引用了一个有效的Material对象，并且该Material对象可以应用光照，则会应用光照模型。Material的如下方法可以用来启用或停用光照：

```
void setLightingEnable(Boolean enable)
```

在这种模式下，对象的颜色绘制由光源和可视对象的相互作用决定。

2. 如果存在顶点颜色描述并且没有被忽略，那么它们将用于绘制多边形。顶点颜色的启用与否由RenderingAttributes对象控制，如下的方法可以设置该状态：

```
void setIgnoreVertexColors(Boolean ignore)
```

当启用顶点颜色时，多边形的明暗处理方案由ColoringAttributes对象决定。平面明暗处理（flat shading）方案对多边形使用单一颜色，而Gouraud明暗处理（Gouraud shading）方案会根据顶点颜色进行插值，确定多边形内部颜色。

3. 如果没有启动光照，同时没有提供顶点颜色（或者被忽略），将使用ColoringAttributes对象的描述进行着色。

程序清单6-7是一个定义了颜色的四面体类。清单6-8则演示了不同外观属性的效果，主窗口中显示了一个旋转的四面体，四面体的外观可通过“Polygon Mode”按钮选择不同的绘制模式：Point、Line和Polygon，还可以通过“Coloring Attribute”按钮选择着色选项：Single、Flat、Gouraud和Lighting（参见图6-23）。

程序清单6-7 ColorTetrahedron.java

```
1 package chapter6;
2
3 import javax.vecmath.*;
4 import javax.media.j3d.*;
5 //定义ColorTetrahedron类，继承自IndexedTriangleArray类
6 public class ColorTetrahedron extends IndexedTriangleArray{
7     public ColorTetrahedron() {
8         super(4, TriangleArray.COORDINATES | TriangleArray.NORMALS |
9             GeometryArray.COLOR_3, 12); //调用父类构造函数进行构造
10        setCoordinate(0, new Point3f(1f,1f,1f)); //设定顶点坐标值
11        setCoordinate(1, new Point3f(1f,-1f,-1f));
12        setCoordinate(2, new Point3f(-1f,1f,-1f));
13        setCoordinate(3, new Point3f(-1f,-1f,1f));
14        int[] coords = {0,1,2,0,3,1,1,3,2,2,3,0};
15        float n = (float)(1.0/Math.sqrt(3));
16        setNormal(0, new Vector3f(n,n,-n)); //设定法向量
17        setNormal(1, new Vector3f(n,-n,n));
18        setNormal(2, new Vector3f(-n,-n,-n));
19    }
20 }
```



```

19    setNormal(3, new Vector3f(-n,n,n));
20    int[] norms = {0,0,0,1,1,1,2,2,2,3,3,3};
21    setCoordinateIndices(0, coords);           //设定顶点坐标索引
22    setNormalIndices(0, norms);               //设定顶点法向量索引
23    setColor(0, new Color3f(1f, 0f, 0f));     //设定颜色
24    setColor(1, new Color3f(0f, 1f, 0f));
25    setColor(2, new Color3f(0f, 0f, 1f));
26    setColor(3, new Color3f(1f, 1f, 1f));
27    setColorIndices(0, coords);               //设定顶点颜色索引
28 }
29 }

```

199

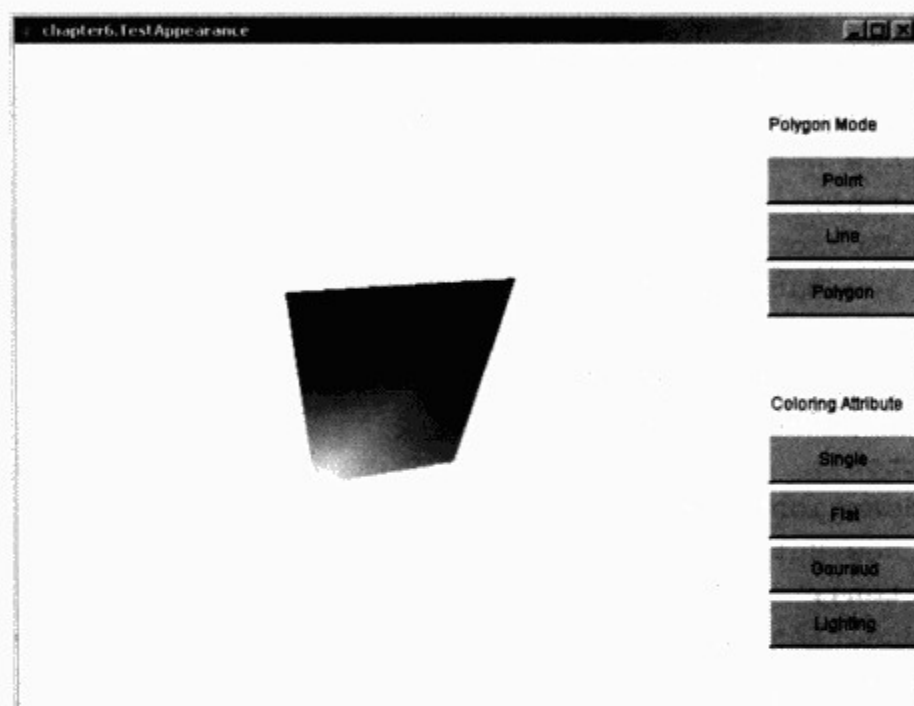


图6-23 不同外观属性下的四面体绘制过程

程序清单6-8 TestAppearance.java

```

1 package chapter6;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义TestAppearance类,继承自Applet类,实现ActionListener接口
12 public class TestAppearance extends Applet implements
13     ActionListener {
14     public static void main(String[] args) {
15         new MainFrame(new TestAppearance(), 640, 480); //创建主窗口,设定窗口大小
16     }
17     //重写Applet的初始化函数
18     public void init() {
19         setLayout(new BorderLayout()); //设置布局
20         Panel p = new Panel();
21         p.setLayout(new GridLayout(12,1,10,5));

```



```

22     add(p, BorderLayout.EAST);
23     p.add(new Panel());
24     p.add(new Label("Polygon Mode"));
25     Button button = new Button("Point");           //添加各按钮及事件侦听器
26     p.add(button);
27     button.addActionListener(this);
28     button = new Button("Line");
29     p.add(button);
30     button.addActionListener(this);
31     button = new Button("Polygon");
32     p.add(button);
33     button.addActionListener(this);
34
35     p.add(new Panel());
36     p.add(new Label("Coloring Attribute"));
37     button = new Button("Single");
38     p.add(button);
39     button.addActionListener(this);
40     button = new Button("Flat");
41     p.add(button);
42     button.addActionListener(this);
43     button = new Button("Gouraud");
44     p.add(button);
45     button.addActionListener(this);
46     button = new Button("Lighting");
47     p.add(button);
48     button.addActionListener(this);
49     //创建Canvas3D对象
50     GraphicsConfiguration gc =
51     SimpleUniverse.getPreferredConfiguration();
52     Canvas3D cv = new Canvas3D(gc);
53     add(cv, BorderLayout.CENTER);
54     BranchGroup bg = createSceneGraph();
55     bg.compile();
56     SimpleUniverse su = new SimpleUniverse(cv); //创建设置SimpleUniverse对象
57     su.getViewingPlatform().setNominalViewingTransform();
58     su.addBranchGraph(bg);
59 }
60
61 Appearance ap;                               //声明Appearance变量
62 private BranchGroup createSceneGraph() { //生成BranchGroup的私有方法
63     BranchGroup root = new BranchGroup();
64     TransformGroup spin = new TransformGroup();
65     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
66     root.addChild(spin);
67
68     //设置允许外观的各种改变
69     ap = new Appearance(); //初始化Appearance变量
70     ap.setCapability(Appearance.ALLOW_COLORING_ATTRIBUTES_WRITE);
71     ap.setCapability(Appearance.ALLOW_POINT_ATTRIBUTES_WRITE);
72     ap.setCapability(Appearance.ALLOW_LINE_ATTRIBUTES_WRITE);
73     ap.setCapability(Appearance.ALLOW_POLYGON_ATTRIBUTES_WRITE);
74     ap.setCapability(Appearance.ALLOW_RENDERING_ATTRIBUTES_WRITE);
75     ap.setCapability(Appearance.ALLOW_MATERIAL_WRITE);

```

200


```
76 Shape3D shape = new Shape3D(new ColorTetrahedron(), ap);
77 //设置形体的旋转变换
78 Transform3D tr = new Transform3D();
79 tr.setScale(0.25);
80 TransformGroup tg = new TransformGroup(tr);
81 spin.addChild(tg);
82 tg.addChild(shape);
83
84 Alpha alpha = new Alpha(-1, 4000);
85 RotationInterpolator rotator =
86 new RotationInterpolator(alpha, spin);
87 BoundingSphere bounds = new BoundingSphere();
88 rotator.setSchedulingBounds(bounds);
89 spin.addChild(rotator);
90
91 //设置背景和光照
92 Background background = new Background(1f, 1f, 1f);
93 background.setApplicationBounds(bounds);
94 root.addChild(background);
95 AmbientLight light = new AmbientLight
96 (true, new Color3f(Color.red));
97 light.setInfluencingBounds(bounds);
98 root.addChild(light);
99 PointLight ptlight = new PointLight(new Color3f(Color.cyan),
100 new Point3f(3f,3f,3f), new Point3f(1f,0f,0f));
101 ptlight.setInfluencingBounds(bounds);
102 root.addChild(ptlight);
103 return root;
104 }
105 //按钮事件响应处理
106 public void actionPerformed(ActionEvent actionEvent) {
107 String cmd = actionEvent.getActionCommand();
108 if ("Point".equals(cmd)) {
109 ap.setPolygonAttributes(new PolygonAttributes(
110 PolygonAttributes.POLYGON_POINT,
111 PolygonAttributes.CULL_BACK,0));
112 ap.setPointAttributes(new PointAttributes(10, false));
113 } else if ("Line".equals(cmd)) {
114 ap.setPolygonAttributes(new PolygonAttributes(
115 PolygonAttributes.POLYGON_LINE,
116 PolygonAttributes.CULL_BACK,0));
117 ap.setLineAttributes(new LineAttributes(3,
118 LineAttributes.PATTERN_DASH, false));
119 } else if ("Polygon".equals(cmd)) {
120 ap.setPolygonAttributes(new PolygonAttributes(
121 PolygonAttributes.POLYGON_FILL,
122 PolygonAttributes.CULL_BACK,0));
123 } else if ("Single".equals(cmd)) {
124 ColoringAttributes ca = new ColoringAttributes();
125 ca.setColor(0.5f, 0.5f, 0.5f);
126 ap.setColoringAttributes(ca);
127 ap.setMaterial(null);
128 RenderingAttributes ra = new RenderingAttributes();
129 ra.setIgnoreVertexColors(true);
```



```

130     ap.setRenderingAttributes(ra);
131 } else if ("Flat".equals(cmd)) {
132     ColoringAttributes ca = new ColoringAttributes();
133     ca.setShadeModel(ColoringAttributes.SHADE_FLAT);
134     ap.setColoringAttributes(ca);
135     ap.setMaterial(null);
136     RenderingAttributes ra = new RenderingAttributes();
137     ra.setIgnoreVertexColors(false);
138     ap.setRenderingAttributes(ra);
139 } else if ("Gouraud".equals(cmd)) {
140     ColoringAttributes ca = new ColoringAttributes();
141     ca.setShadeModel(ColoringAttributes.SHADE_GOURAUD);
142     ap.setColoringAttributes(ca);
143     ap.setMaterial(null);
144     RenderingAttributes ra = new RenderingAttributes();
145     ra.setIgnoreVertexColors(false);
146     ap.setRenderingAttributes(ra);
147 } else if ("Lighting".equals(cmd)) {
148     ap.setMaterial(new Material());
149     RenderingAttributes ra = new RenderingAttributes();
150     ra.setIgnoreVertexColors(true);
151     ap.setRenderingAttributes(ra);
152 }
153 }
154 }

```

202

ColorTetrahedron类与程序清单6-1中定义的类相似，它加入了顶点颜色定义（第23到26行）。TestAppearance程序创建了一个ColorTetrahedron实例，并用做一个Shape3D叶节点的几何属性，程序中还为这个几何体创建了一个Appearance对象，并对该对象设置了多个允许标志位：

```

ap.setCapability(Appearance.ALLOW_COLORING_ATTRIBUTES_WRITE);
ap.setCapability(Appearance.ALLOW_POINT_ATTRIBUTES_WRITE);
ap.setCapability(Appearance.ALLOW_LINE_ATTRIBUTES_WRITE);
ap.setCapability(Appearance.ALLOW_POLYGON_ATTRIBUTES_WRITE);
ap.setCapability(Appearance.ALLOW_RENDERING_ATTRIBUTES_WRITE);
ap.setCapability(Appearance.ALLOW_MATERIAL_WRITE);

```

这些设置允许对Appearance对象的颜色属性、点属性、线属性、多边形属性及材质属性进行动态修改。值得注意的是，这些允许标志位的设置必须单独进行，不可以在一次方法调用中组合多个位。

程序通过按钮来控制外观设置（第106行），按钮可分为两类，分别控制多边形模式和着色属性，共有7个按钮。

Point：设置PolygonAttributes为POLYGON_POINT，只绘制顶点，Appearance对象的PointAttributes设置顶点的大小为10。

Line：设置PolygonAttributes为POLYGON_LINE，绘制对象的线框，Appearance对象的LineAttributes设置线型为虚线（dash），线宽为3。

Polygon：设置PolygonAttributes为POLYGON_FILL，绘制对象的面。

Single：通过ColoringAttributes设置物体的绘制颜色为单一的灰色，设置Appearance对象的材质属性为空，以关闭光照效果，设置RenderingAttributes对象忽略顶点颜色。

Flat：根据顶点数据，以纯色绘制几何对象。设置Appearance对象的材质属性为空以关闭光照，设置ColoringAttributes对象选择平面明暗处理模式。

Gouraud: 基于Gouraud明暗处理方法, 用各顶点的颜色来绘制几何对象。设置Appearance对象的材质属性为空以关闭光照, 设置ColoringAttributes对象选择Gouraud明暗处理模式。

Lighting: 物体有光源进行照射, 设置Appearance对象的Material属性为默认对象, 设置RenderingAttributes对象以忽略顶点颜色。

203

点、线和多边形的绘制模式的选择, 独立于single、flat、Gouraud和光照着色的选择。因此, 即使在只绘制顶点的多边形绘制模式下, 也可使用光照。

如果在程序中放置一个Dodecahedron对象, 将看到在线框形式下, 除了五边形的边以外, 还有其他的边存在。这是由GeometryInfo类的自动三角化造成的, 这些五边形被分割成多个三角形。

主要的类和方法

- `javax.vecmath.Point*` 封装了几何点的类族。
- `javax.vecmath.Vectir*` 封装了几何向量的类族。
- `javax.vecmath.Color*` 颜色值类族。
- `javax.media.j3d.Shape3D` 封装了可视对象的叶节点类。
- `javax.media.j3d.Geometry` 为Shape3D对象定义几何属性的节点组件类。
- `javax.media.j3d.Appearance` 为Shape3D对象定义外观属性的节点组件类。
- `javax.media.j3d.GeometryArray` 根据顶点描述构造几何体的类族。
- `javax.media.j3d.GeometryStripArray` 构造带状几何体的GeometryArray类族。
- `javax.media.j3d.IndexedGeometryArray` 根据顶点和索引定义构造几何体的GeometryArray类族。
- `javax.media.j3d.IndexedGeometryStripArray` 根据顶点和索引描述构造带状几何体的GeometryArray类族。
- `javax.media.j3d.GeometryInfo` 构造和操作几何体的类。
- `javax.media.j3d.Primitive` 几何基元类, 派生自Group类。
- `javax.media.j3d.ColoringAttributes` 用于描述颜色和明暗处理的节点组件类。
- `javax.media.j3d.TransparencyAttributes` 用于表示透明度的节点组件类。
- `javax.media.j3d.PointAttributes` 表示点绘制属性(大小、反走样等)的节点组件类。
- `javax.media.j3d.LineAttributes` 表示线绘制属性(线宽、线型、反走样等)的节点组件类。
- `javax.media.j3d.PolygonAttributes` 表示多边形绘制属性(光栅化模式、背向面去除、偏移等)的节点组件类。
- `javax.media.j3d.Font3D` 封装3D字体的类。
- `javax.media.j3d.Text3D` 表示3D文本字符串的几何体类。
- `javax.media.j3d.Text2D` 在3D空间中表示2D文本字符串的类, 派生自Shape3D类。
- `javax.media.j3d.FontExtrusion` 定义了从Font对象生成Font3D对象的凸出路径(extrusion path)的类。

关键术语

- **点 (point)** 基本的几何元素, 表示空间中的一个位置。
- **向量 (vector)** 表示具有方向性的量的几何元素。
- **多边形网格 (polygon mesh)** 一系列简单的多边形(如三角形), 用于拟合表示一个曲面。
- **柏拉图实体 (Platonic solids)** 五种正多面体: 正四面体、正六面体(立方体)、正八面体、正十二面体和正二十面体。
- **平面明暗处理 (flat shading)** 一种简单地将一个多边形片用单一颜色着色的方案。

204

- **Gouraud明暗处理** (Gouraud shading) 一种根据顶点颜色的插值对内部点进行着色的方案。

本章提要

- 在本章中，我们讨论了通过几何属性和外观属性的定义，来构造可视对象的基础知识。几何属性定义了对对象的形状和大小，外观属性定义了颜色、材质和纹理等绘制属性。
- Java 3D编程库中包含了javax.vecmath包，其中定义了大量的点、向量和矩阵相关类，它们在Java 3D的其他部分用于描述几何属性、几何变换及其他一些属性。
- Shape3D类表示场景图中的一个可视对象，Shape3D引用Geometry和Appearance对象来定义其几何属性和外观属性。
- Java 3D为定义几何体提供了大量的工具类。GeometryArray及其派生类可通过直接指定低层的点、线和简单多边形来定义几何体。
- GeometryInfo类允许使用更一般的多边形，它使用NormalGenerator类辅助自动生成法向量，使用Stripifier附着自动生成几何带。
- 几何基元是预定义的高层次几何对象，Java 3D工具包中实现了一些常用的几何基元，包括长方体、圆锥体、圆柱体及球体。
- 文本可当做特殊的几何对象。Text3D是Geometry的子类，它通过关联Font3D、FontExtrusion和Font类对象，定义了3D立体文本对象。Text2D是Shape3D的子类，它将2D文本实现为经过纹理映射的矩形。
- 形状的外观属性由多种以Appearance节点为根的节点组件定义。一个Appearance节点会引用其他节点组件类，如ColoringAttributes、PointAttributes、LineAttributes、PolygonAttributes、TransparencyAttributes和Material。

复习题

6.1 构造Point3f对象表示如下点：

(1, 2, 3), (0, 0, 0), (-1.2, 3.4, -5.6)

205 6.2 写出计算习题6.1中所定义的点之间的三个距离的Java语句。

6.3 构造Vector3f对象以表示如下向量：

(1, 2, 3), (1, 1, 1), (-1.2, 3.4, -5.6)

6.4 写出计算习题6.3中所定义的向量之间的三个夹角的Java语句。

6.5 构造一个PointArray对象，以表示一个中心在原点的立方体的所有顶点，立方体的边长为1.0，边平行于坐标轴。

6.6 构造一个LineArray对象，以表示一个正四面体的边。

6.7 使用TriangleArray对象表示一个正四面体的几何特征。

6.8 使用QuadArray表示一个立方体的几何要素。

6.9 构造一个LineStripArray对象以表示一个正四面体的边。

6.10 构造一个TriangleStripArray对象以表示一个正四面体的几何特征。

6.11 构造一个TriangleFanStripArray对象以表示一个正四面体的几何特征。

6.12 使用一个IndexedQuadArray对象表示一个立方体的几何特征。

6.13 画出如下TriangleStripArray对象所定义的几何体：

```
int[] stripVertexCounts = {5, 3};
TriangleStripArray tsa = new TriangleStripArray
    (8, GeometryArray.COORDINATES, stripVertexCounts);
Point3f[] coords = new Point3f[8];
```



```

coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(0f, 1f, 0f);
coords[2] = new Point3f(0.5f, 0.866f, 0f);
coords[3] = new Point3f(1.5f, 0.866f, 0f);
coords[4] = new Point3f(1f, 1.73f, 0f);
coords[5] = new Point3f(0f, 1f, 0f);
coords[6] = new Point3f(1.5f, 0.866f, 0f);
coords[7] = new Point3f(2f, 0f, 0f);
tsa.setCoordinates(0, coords);

```

6.14 画出如下TriangleFanStripArray对象所定义的几何体:

```

int[] stripVertexCounts = {4, 4};
TriangleFanArray tfa = new TriangleFanArray
    (8, GeometryArray.COORDINATES, stripVertexCounts);
Point3f[] coords = new Point3f[8];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(1f, 0f, 0f);
coords[2] = new Point3f(0.866f, 0.5f, 0f);
coords[3] = new Point3f(0.5f, 0.866f, 0f);
coords[4] = new Point3f(0f, 0f, 0f);
coords[5] = new Point3f(-1f, 0f, 0f);
coords[6] = new Point3f(-0.866f, -0.5f, 0f);
coords[7] = new Point3f(-0.5f, -0.866f, 0f);
tfa.setCoordinates(0, coords);

```

6.15 画出如下IndexedTriangleFanStripArray对象所定义的几何体:

```

int[] stripIndexCounts = {5, 3};
IndexedTriangleStripArray itsa = new
    IndexedTriangleStripArray(6,
        GeometryArray.COORDINATES, 8, stripIndexCounts);
Point3f[] coords = new Point3f[6];
coords[0] = new Point3f(0f, 0f, 0f);
coords[1] = new Point3f(0f, 1f, 0f);
coords[2] = new Point3f(0.5f, 0.866f, 0f);
coords[3] = new Point3f(1.5f, 0.866f, 0f);
coords[4] = new Point3f(1f, 1.73f, 0f);
coords[5] = new Point3f(2f, 0f, 0f);
itsa.setCoordinates(0, coords);
int[] indices = {0, 1, 2, 3, 4, 1, 3, 5};
itsa.setCoordinateIndices(0, indices);

```

206

6.16 使用GeometryInfo对象定义一个边长为1的立方体。

6.17 Box类和ColorCube类的区别是什么?

6.18 Text3D和Text2D除了维度以外,还有什么不同?

6.19 如果一个几何体同时定义了顶点坐标和顶点颜色,如何使用这个几何体并忽略顶点的颜色?

6.20 推导出如下参数方程所确定的莫比乌斯(Möbius)带的曲面法向量的方程:

$$\begin{aligned}
 x &= (1+v\cos(u/2))\cos u \\
 y &= (1+v\cos(u/2))\sin u \\
 z &= v\sin(u/2) \\
 0 &\leq u \leq 2\pi, -0.3 \leq v \leq 0.3
 \end{aligned}$$

编程练习

6.1 正八面体是五种柏拉图实体中的一种,它有八个三角形面和6个顶点:

(0, 0, 1)
 (-1, 0, 0), (0, -1, 0), (1, 0, 0), (0, 1, 0)
 (0, 0, -1)

请实现Octahedron类, 该类是Geometry的一个子类。

6.2 请编写Octahedron类的测试程序, 以线框形式显示八面体。

6.3 正二十面体 (Icosahedron) 是另一种柏拉图实体, 其顶点如下:

(0, ϕ , 1)
 (1, 0, ϕ), (-1, 0, ϕ), ($-\phi$, 1, 0), (0, ϕ , -1), (ϕ , 1, 0),
 (0, $-\phi$, 1), ($-\phi$, -1, 0), (-1, 0, $-\phi$), (1, 0, $-\phi$), (ϕ , -1, 0),
 (0, $-\phi$, -1)

其中 $\phi = (\sqrt{5} + 1)/2$ 。使用GeometryInfo实现Icosahedron类 (作为Shape3D的子类), 并使用NormalGenerator类来生成法向量。

207 6.4 为题6.3中所定义的二十面体编写测试程序, 使用光照并旋转该几何对象。

6.5 使用LineArray类创建并显示 $3D4 \times 4 \times 4$ 网格。

6.6 使用IndexedQuadArray对象定义一个平截体 (顶部正方形大小为 1×1 , 底部正方形大小为 2×2 , 高度为1), 并在虚拟空间中旋转该平截体。

6.7 使用GeometryInfo类创建并显示一个金字塔体。

6.8 将程序清单6-8中的正四面体换成一个正十二面体, 并在线框形式下观察GeometryInfo类的三角形化结果。

6.9 修改程序清单5-1以支持3D文本的动态改变, 实现一个菜单项和一个对话框让用户输入字符串, 并将字符串以3D文本的形式显示出来。

6.10 编写类似于题6.9的Java 3D程序, 但以Text2D对象显示字符串。

6.11 扩展程序清单6-8的功能, 以支持不同种类的线型。

6.12 实现ColorOctahedron类, 为不同顶点指定不同颜色, 并编写测试程序, 以Gouraud shading方式显示ColorOctahedron对象。

6.13 使用GeometryInfo创建一个复习题6.20中定义的莫比乌斯 (Möbius) 带, 用NormalGenerator类生成法向量, 并在光照情况下显示该莫比乌斯带。

208 6.14 使用IndexedQuadArray对象创建一个Möbius带, 基于复习题6.20所推导的公式设置其法向量, 并在光照情况下显示该莫比乌斯带。

第7章 几何变换

学习目标

- 描述和3D图形有关的变换。
- 建立包括平移、旋转、缩放、错切和反射变换在内的3D仿射变换。
- 理解并应用变换矩阵。
- 在场景图中应用变换。
- 建立并应用复合变换。
- 在几何体构造中应用变换。

209

7.1 引言

几何变换在计算机图形学中起着至关重要的作用。通过适当的几何变换，可以改变图形对象的形状、尺寸及位置，以完成对它们的定义。仿射变换经常用于虚拟世界的建模，在观察过程中也同样重要。透视或正交投影（perspective or orthogonal projection）是一种特殊的变换，通常用于视图变换中实现3D空间到2D图像的映射。动画通常需要应用几何变换，以得到场景中的动态效果。

Java 3D在多个层次提供了变换，像Matrix4d这样的矩阵类提供了变换的低层数据表示。Transform3D类封装了3D变换，便于设置和组合变换，以及将变换应用于点和向量。TransformGroup类表示场景图中高层次的变换节点。

通过齐次坐标的使用，所有的3D投影变换（包含仿射变换）由 4×4 的变换矩阵完全决定。像旋转和平移这样具有直接几何解释的变换，可以用矩阵的形式表达。但是，某些变换，例如一般3D旋转，显式地构造它们的矩阵可能会比较复杂。其他的表示方法，例如应用于3D旋转的四元数表示法，可以为构造和操作提供更方便的中间形式。

变换可以组合起来形成新的变换，简单变换的组合是构造和操纵复杂变换的有效方法。在Java 3D中，通过变换的相乘，复合变换可以发生在低层的矩阵对象或Transform3D对象之上。它也可以发生在较高层次的场景图中，在这里，一串变换节点链形成了复合变换的效果。

变换的主要应用就是改变对象的几何特征。在一个Java 3D场景图中，TransformGroup节点代表变换，一个TransformGroup节点将施加由其子节点定义的变换。从一个对象到Locale节点之间的一系列TransformGroup节点，可以针对对象施加多种方式的变换。场景图的类似树形结构为用变换节点操纵可视对象提供了很大的灵活性。

变换的另一个应用是辅助几何体的构造。Transform3D类提供了很多方便的方法，来对点和向量应用变换。为了构造具有某些对称特性的几何体，我们可以利用变换的能力，通过对一些基点施加变换，来产生几何体的其他许多点。

7.2 3D仿射变换

变换是从一个向量空间到它本身或到另一个向量空间的映射。保持某些几何特性的一些变换族，在计算机图形学中有着特殊意义。例如，投影变换在3D观察中起着至关重要的作用。作为投影变换的子集，仿射变换广泛用于可视对象的建模。

仿射变换将直线映射成直线并且保持平行性。例如，仿射变换并不一定能将矩形映射成矩形，但它总是将矩形映射成平行四边形。类似于2D的情形，基本的3D仿射变换包括平移、旋转、缩放、错切和反射，但3D情形通常更复杂，尤其是旋转变换。

保持任意两点间距离不变的仿射变换，称为刚性运动、欧几里得运动或等距变换 (isometry)。平移变换、旋转变换和反射变换都是刚性运动的例子。

7.2.1 变换矩阵

仿射变换可以用矩阵形式来表示。如果3D空间中的一个点用3个坐标表示：

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

那么仿射变换可以用下列矩阵方程写出：

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

相对于原点的缩放变换和错切变换，可以只用矩阵乘法来表示。在变换中加上列向量 b 则表示平移。由于需要对平移进行特殊处理，以上的变换在 R^3 空间中不是线性的。但是，若利用齐次坐标来表示3D点，则所有的仿射变换和投影变换都可以表示成矩阵乘法。3D点的齐次坐标有四个组成部分： (x, y, z, w) 。当 w 不等于0时，它相当于一般3D坐标 $(x/w, y/w, z/w)$ 。利用齐次坐标，上面给出的变换可以等价地表示为一个 R^4 空间中的线性变换

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

以上方程中的 4×4 变换矩阵完全决定了一个仿射变换，变换操作可以表示为相应的矩阵运算。若干个仿射变换的组合，可以产生另一个仿射变换，复合变换的矩阵就是相应变换矩阵的积，变换矩阵的逆矩阵对应于逆变换。更为一般地，一个投影变换可以用类似的矩阵方程来表示：

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

关于矩阵及其运算的介绍，可以参见附录A。

Java 3D提供的许多类支持变换。除了向量类，`javax.vecmath`包还含有表示 3×3 、 4×4 以及一般矩阵的矩阵类：`Matrix3f`、`Matrix3d`、`Matrix4f`、`Matrix4d`与`GMatrix`。以下的代码片段构造了一个`Matrix4d`对象：

```
double[] array = {1.0 2.0 3.0 1.0,
                  0.0 1.0 -1.0 2.0,
                  4.0 0.0 0.5 -1.0,
                  0.0 0.0 0.0 1.0};
```

```
Matrix4d matrix = new Matrix4d(array);
```


GMatrix类可以表示任意大小的double类型矩阵。例如，下面的代码生成了一个 3×4 矩阵：

```
double[] array = {1.0  2.0  3.0  1.0,
                  0.0  1.0 -1.0  2.0,
                  4.0  0.0  0.5 -1.0};
```

```
GMatrix matrix = new GMatrix(3, 4, array);
```

它支持例如加法、乘法和求逆这样的基本矩阵运算。以下是Matrix4d类关于矩阵运算的部分方法列表，其他类也包含了类似的方法。

- **void** add(Matrix4d m1): 将矩阵m1加到当前矩阵。
- **void** sub(Matrix4d m1): 将当前矩阵减去矩阵m1。
- **void** mul(Matrix4d m1): 将当前矩阵乘以矩阵m1。
- **void** invert(): 对当前矩阵求逆。
- **void** add(Matrix4d m1, Matrix4d m2): 将当前矩阵设为矩阵m1和m2的和。
- **void** sub(Matrix4d m1, Matrix4d m2): 将当前矩阵设为矩阵m1和m2的差。
- **void** mul(Matrix4d m1, Matrix4d m2): 将当前矩阵设为矩阵m1和m2的积。
- **void** invert(Matrix4d m1): 将当前矩阵设为矩阵m1的逆。
- **void** transpose(): 将当前矩阵转置。
- **void** mul(double scalar): 将当前矩阵乘以数scalar。
- **double** determinant(): 求出当前矩阵的行列式值。

程序清单7-1给出了一个用于显示矩阵的类，程序清单7-2利用Matrix4d类的方法，提供了一个交互式的用户界面，用于观察矩阵以及执行矩阵上的各种运算。

程序清单7-1 MatrixPanel.java

```
1 package chapter7;
2
3 import java.awt.*;
4 import javax.vecmath.*;
5 //定义MatrixPanel类，继承自Panel类
6 public class MatrixPanel extends Panel {
7     TextField[] fields = new TextField[16];    //定义并创建文本字段数组
8
9     public MatrixPanel() { //默认无参构造函数，构造单位矩阵
10         setLayout(new GridLayout(4, 4)); //设置布局
11         for (int i = 0; i < 16; i++) { //初始化各文本字段，构造单位矩阵
12             fields[i] = new TextField(5);
13             if (i/4 == i%4)
14                 fields[i].setText("1");
15             else
16                 fields[i].setText("0");
17             add(fields[i]);
18         }
19     }
20
21     public MatrixPanel(Matrix4d m) { //用矩阵m作参数的构造函数
22         setLayout(new GridLayout(4, 4)); //设置布局
23         for (int i = 0; i < 16; i++) {
24             fields[i] = new TextField(5);
25             fields[i].setText("" + m.getElement(i/4, i%4));
```

```

26     add(fields[i]);
27 }
28 }
29
30 public void set(Matrix4d m) { //根据矩阵m设置各文本字段
31     for (int i = 0; i < 16; i++) {
32         fields[i].setText("" + m.getElement(i/4, i%4));
33     }
34 }
35
36 public void get(Matrix4d m) { //从各文本字段获取数值并存放在矩阵m中
37     for (int i = 0; i < 16; i++) {
38         m.setElement(i/4, i%4, Double.parseDouble
39             (fields[i].getText()));
40     }
41 }
42 }

```

程序清单7-2 TestMatrix.java

```

1 package chapter7;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.vecmath.*;
6 import java.applet.*;
7 import com.sun.j3d.utils.applet.MainFrame;
8 //定义TestMatrix类, 继承自Applet类, 实现ActionListener接口
9 public class TestMatrix extends Applet implements ActionListener {
10     public static void main(String[] args) {
11         new MainFrame(new TestMatrix(), 600, 200); //创建主窗口, 设置窗口大小
12     }
13
14     MatrixPanel mp; //声明MatrixPanel类型变量
15     Matrix4d m = new Matrix4d(); //声明并创建Matrix4d对象
16     TextField tf; //声明文本字段变量
17     //重写Applet初始化函数
18     public void init() {
19         this.setLayout(new BorderLayout()); //设置布局管理器为BorderLayout
20
21         mp = new MatrixPanel(); //初始化MatrixPanel变量
22         add(mp, BorderLayout.CENTER);
23
24         Panel p = new Panel(); //创建放置按钮的Panel对象并设置布局
25         p.setLayout(new GridLayout(6,1));
26         Button button = new Button("Identity"); //添加各按钮及事件侦听器
27         button.addActionListener(this);
28         p.add(button);
29         button = new Button("Zero");
30         button.addActionListener(this);
31         p.add(button);
32         button = new Button("Negate");
33         button.addActionListener(this);
34         p.add(button);

```



```
35     button = new Button("Transpose");
36     button.addActionListener(this);
37     p.add(button);
38     button = new Button("Invert");
39     button.addActionListener(this);
40     p.add(button);
41     button = new Button("Determinant");
42     button.addActionListener(this);
43     p.add(button);
44     this.add(p, BorderLayout.EAST);
45
46     tf = new TextField();           //初始化文本字段变量
47     add(tf, BorderLayout.SOUTH);
48 }
49 //事件处理函数
50 public void actionPerformed(ActionEvent e) {
51     String cmd = e.getActionCommand();
52     if ("Identity".equals(cmd)) { //设为单位矩阵
53         mp.get(m);
54         m.setIdentity();
55         mp.set(m);
56     } else if ("Zero".equals(cmd)) { //设为零矩阵
57         mp.get(m);
58         m.setZero();
59         mp.set(m);
60     } else if ("Negate".equals(cmd)) { //设为负矩阵
61         mp.get(m);
62         m.negate();
63         mp.set(m);
64     } else if ("Transpose".equals(cmd)) { //设为转置矩阵
65         mp.get(m);
66         m.transpose();
67         mp.set(m);
68     } else if ("Invert".equals(cmd)) { //设为逆矩阵
69         mp.get(m);
70         m.invert();
71         mp.set(m);
72     } else if ("Determinant".equals(cmd)) { //得到行列式的值并显示
73         mp.get(m);
74         tf.setText("" + m.determinant());
75     }
76 }
77 }
```

MatrixPanel类是一个用来显示 4×4 矩阵的可视组件，它为矩阵的16个元素提供了16个文本字段（第7行），它的一个构造函数用Matrix4d对象作为参数的初始显示值，它的默认构造函数将显示初始化为一个单位矩阵（第9行）。set (Matrix4d m) 方法将显示设为所给的矩阵，get (Matrix4d m) 方法可以获得显示中的矩阵。

TestMatrix类是主测试程序，它生成了一个MatrixPanel对象，用于表示一个Matrix4d对象（第21行，见图7-1）。在窗口的EAST区安放了6个按钮，用于执行Matrix4d类的方法所定义的相关矩阵运算。

- Identity：利用identity()方法将矩阵设置为单位矩阵。

214

- Zero: 利用zero()方法将矩阵设置为零矩阵。
- Negative: 利用negate()方法将矩阵设置为它的负矩阵。
- Transpose: 利用transpose()方法将矩阵设置为它的转置矩阵。
- Invert: 利用invert()方法对矩阵求逆, 并显示其逆矩阵。
- Determinant: 利用determinant()方法计算矩阵的行列式值, 并将结果显示在底部的文本字段之中。

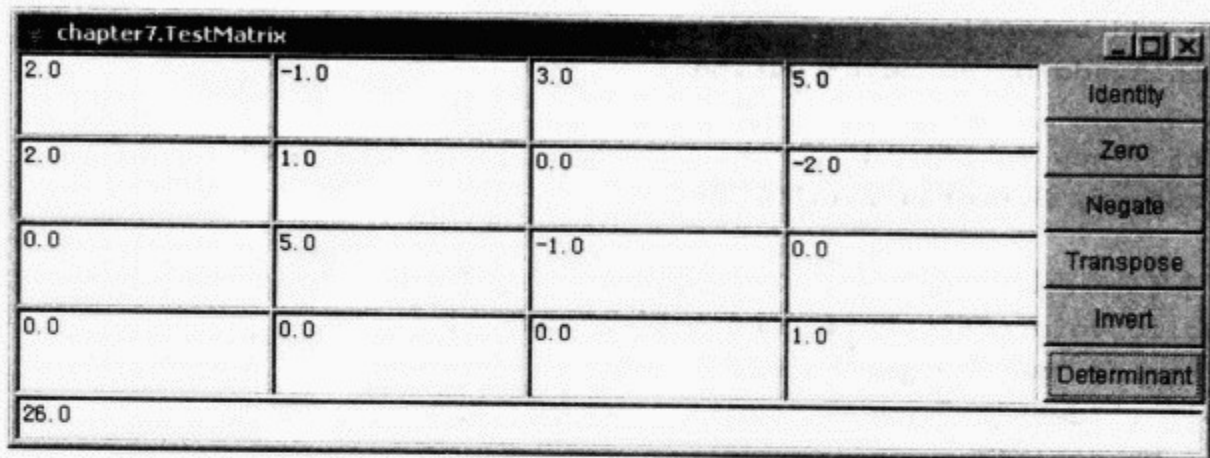


图7-1 矩阵运算的测试程序。变换矩阵显示于文本字段网格中, 用户可以直接编辑输入值, 程序提供了6种运算, 行列式的计算结果显示在底部的文本字段之中

7.2.2 类Transform3D

Java 3D包含了Transform3D类来表示3D仿射变换或投影变换。在场景图中, 一个TransformGroup节点利用Transform3D对象来定义其变换, Transform3D在其内部保留了一个 4×4 的double型矩阵以表示其变换。要生成一个Transform3D对象, 需要用到它的某一个构造函数, 默认的构造函数生成一个单位矩阵。可以为变换矩阵的构造函数提供一个矩阵对象、一个数组或其他形式的参数。例如, 下面三个Transform3D对象表示相同的变换:

```
double[] array = {1.0 2.0 3.0 1.0,
                  0.0 1.0 -1.0 2.0,
                  4.0 0.0 0.5 -1.0,
                  0.0 0.0 0.0 1.0};

Matrix4d matrix = new Matrix4d(array);
GMatrix gmatrix = new GMatrix(4,4,array);
Transform3D transform1 = new Transform3D(matrix);
Transform3D transform2 = new Transform3D(gmatrix);
Transform3D transform3 = new Transform3D(array);
```

Transform3D包含了大量方便的方法, 用于建立和操作变换, 其中一些直接操作变换矩阵的方法列表如下:

- void set(Matrix4d m1): 根据矩阵m1设置变换矩阵。
- void set(Matrix4f m1): 根据矩阵m1设置变换矩阵。
- void set(GMatrix m1): 根据矩阵m1设置变换矩阵。
- void set(double[] array): 根据double型数组array设置变换矩阵。
- void set(float[] array): 根据float型数组array设置变换矩阵。
- void get(Matrix4d m1): 获取变换矩阵并赋给Matrix4d型的矩阵m1。
- void get(Matrix4f m1): 获取变换矩阵并赋给Matrix4f型的矩阵m1。

- **void get(GMatrix m1):** 获取变换矩阵并赋给GMatrix型的矩阵m1。
- **void get(double[] array):** 获取变换矩阵并赋给double型数组array。
- **void get(float[] array):** 获取变换矩阵并赋给float型数组array。

变换也可以采用诸如平移、缩放、旋转和错切之类的几何描述来定义。

平移变换用变换矩阵表示为:

$$\begin{bmatrix} 1 & 0 & 0 & b_1 \\ 0 & 1 & 0 & b_2 \\ 0 & 0 & 1 & b_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

在这个平移变换的作用下, 每个点都分别在 x 、 y 和 z 方向移动一个常量 b_1 、 b_2 、 b_3 。很明显, 在一个平移变换作用下, 几何体的形状和方向并不会变化, 由 b_1 、 b_2 、 b_3 决定的平移的逆变换, 就是由 $-b_1$ 、 $-b_2$ 、 $-b_3$ 决定的平移变换。

Transform3D包含了下列方法, 来建立平移变换:

```
void set(Vector3d trans)
void set(Vector3f trans)
void setTranslation(Vector3d trans)
void setTranslation(Vector3f trans)
```

set方法用指定的平移变换代替了整个变换, 而setTranslation方法只是修改了已有变换中的平移部分。

在 x 、 y 、 z 方向上, 缩放因子分别为 s_1 、 s_2 、 s_3 的缩放变换的矩阵表示为:

$$\begin{bmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

如果所有的缩放因子都非零, 那么这个缩放变换是可逆的, 它的逆变换也是一个缩放变换, 缩放因子为 $1/s_1$ 、 $1/s_2$ 、 $1/s_3$ 。如果 $s_1 = s_2 = s_3$, 则这个缩放变换是等距的 (uniform)。

Transform3D包含了下列方法来建立缩放变换:

```
void set(double scale)
void setScale(double scale)
void setScale(Vector3d scales)
```

set方法用指定的缩放变换代替了整个变换, 而setScale方法只是修改了已有变换中的缩放部分。

3D反射变换是关于一个平面进行的, 例如, 一个关于 xy 平面的简单反射变换, 可由以下矩阵给出:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

反射变换总是可逆的, 反射变换的逆就是它本身。更一般地, 关于经过原点且法向量为 u 的平面的对称变换, 可以表示成:

216

$$T(x) = x - \frac{2x \cdot u}{\|u\|^2} u$$

也可以从这个方程中导出对应的矩阵表示。另一种建立这种一般性的反射变换的方法，将在稍后的章节中介绍。

3D错切变换沿着一个平面移动点，而且移动量决定于点到平面的距离。只改变x坐标的错切变换，由下列矩阵表示：

$$\begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

在这个变换下，y坐标和z坐标不会改变，x坐标根据等式 $x' = x + sh_x z$ 进行移动。

更一般的(x, y)错切变换改变了x坐标和y坐标：

$$\begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

错切变换是可逆的，以上错切变换的逆是另一个同类型的错切变换，它的变换参数为 $-sh_x$ 与 $-sh_y$ 。

7.2.3 旋转

3D旋转变换是复合运算。一般的3D旋转变换有一个旋转轴，它可以是空间中的任意直线，围绕这个轴，所有点都旋转一个固定的角度。关于z轴角度为 θ 的旋转变换，可以用下列矩阵表示：

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

虽然任何旋转变换都可以表示成变换矩阵的形式，但要想直接从一般旋转的几何定义中获得变换矩阵，通常是很困难的。例如，可以很容易地得到绕x轴、y轴或z轴的旋转变换的矩阵，但是围绕轴(1,1,1)进行角度为 $\pi/3$ 的旋转变换，旋转变换矩阵是什么呢？一种与一般3D旋转变换的几何特性有更直接联系的表示法，需要用到四元数。四元数(quaternion)是复数域的扩展，关于四元数的介绍，可以参考附录A。

3D空间中的一点(x, y, z)对应一个纯四元数(实数部分为0的四元数)：

$$p = xi + yj + zk$$

令q为一固定的四元数，则3D空间中的变换定义为：

$$T_q(p) = qp\bar{q}$$

如果q是一单位四元数，那么可以验证以上定义的变换是一个旋转变换。在这种情况下，q可以写成：

$$q = \cos \frac{\theta}{2} + u \sin \frac{\theta}{2}, \quad u = ai + bj + ck$$

217

单位向量 u 定义了旋转轴的方向，而且该旋转轴经过原点，旋转的角度由 θ 给定。

Java 3D包含了Quat4f类和Quat4d类用于表示四元数。除了Tuple4*类固有的运算功能之外，四元数类甚至还支持共轭、乘法、求逆和归一化（conjugate, multiply, inverse and normalize）等运算。由于四元数与旋转变换之间存在联系，四元数类甚至提供了一些方便的方法，用于根据指定的旋转轴和旋转角，直接设置旋转四元数：

```
void set(AxisAngle4d r)
```

Transform3D类提供了若干构造函数和方法，可以直接接受四元数参数来定义旋转变换：

```
Transform3D(Quat4d q, Vector3d translation, double scale)
Transform3D(Quat4f q, Vector3d translation, double scale)
Transform3D(Quat4f q, Vector3f translation, float scale)
void set(Quat4d q)
void set(Quat4f q)
```

另一种表示3D旋转的常用方法是利用围绕坐标轴的旋转某角度的三个旋转变换 $R_3R_2R_1$ 。旋转变换 R_2 的旋转轴不同于 R_1 和 R_3 的旋转轴。这三个旋转变换的角度称为欧拉角（Euler angles）。旋转轴可以有不同的旋转，旋转角则有多种叫法——例如，（仰角、方位角、倾斜角）、（转角、下倾角、偏转角）、（摆角、弯角、旋转角）与（方向角、纬度角、坡度角）。Transform3D类提供了一个方法，可以基于欧拉角设置旋转变换：

```
void setEuler(Vector3d eulerAngles)
```

参数Vector3D对象指定了关于x、y、z轴的欧拉角，这些角度也称为坡度角、纬度角、方向角。但是，Tranform3D中没有提供读取旋转变换的欧拉角的方法，可以用一个get方法获取其四元数，然后再把它转化成欧拉角。程序清单7-3中的代码实现了从四元数到欧拉角的转化。

程序清单7-3 quatToEuler

```
1 public static Vector3d quatToEuler(Quat4d q1) { //将四元数转换为欧拉角
2     double sqw = q1.w*q1.w;
3     double sqx = q1.x*q1.x;
4     double sqy = q1.y*q1.y;
5     double sqz = q1.z*q1.z;
6     double heading = Math.atan2(2.0 * (q1.x*q1.y + q1.z*q1.w),
7     (sqx - sqy - sqz + sqw));
8     double bank = Math.atan2(2.0 * (q1.y*q1.z + q1.x*q1.w),
9     (-sqx - sqy + sqz + sqw));
10    double attitude = Math.asin(-2.0 * (q1.x*q1.z - q1.y*q1.w));
11    return new Vector3d(bank, attitude, heading);
12 }
```

Transform3D包含了一些类似于Matrix4d类中的变换矩阵运算的方法，它同样允许对点和向量直接应用变换。这个特性简化了通过变换构造几何体的过程，这方面的内容将在本章稍后进行讨论。

程序清单7-4演示了Transform3D类的功能（见图7-2），程序清单7-5定义类显示一组坐标轴。程序显示了一个和Transform3D对象相应的变换矩阵，而且它可以由用户进行编辑。变换的旋转、平移和缩放部分被提取出来并分别加以显示。用户也可以指定这些部分，将它们直接用于变换。变换应用于一个可视对象——一组3D坐标轴，所以它的效果马上就能看出来。这个程序提供了一个工具，用于探索变换矩阵和变换的几何解释之间的关系（见图7-2）。

程序清单7-4 TestTransform.java

```
1 package chapter7;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义TestTransform类,继承自Applet类,实现ActionListener接口
12 public class TestTransform extends Applet implements
13 ActionListener {
14     public static void main(String[] args) {
15         new MainFrame(new TestTransform(), 640, 300); //创建主窗口,设置窗口大小
16     }
17
18     TransformGroup trGroup;
19     Transform3D transform = new Transform3D();
20     MatrixPanel mp = new MatrixPanel(); //声明并创建MatrixPanel对象
21     TextField rx = new TextField(); //声明并创建多个文本字段
22     TextField ry = new TextField();
23     TextField rz = new TextField();
24     TextField ra = new TextField();
25     TextField tx = new TextField();
26     TextField ty = new TextField();
27     TextField tz = new TextField();
28     TextField sx = new TextField();
29     TextField sy = new TextField();
30     TextField sz = new TextField();
31     //重写Applet初始化函数
32     public void init() {
33         setLayout(new BorderLayout()); //设置界面布局为BorderLayout
34
35         Panel eastPanel = new Panel(); //创建Panel对象eastPanel作为容器
36         eastPanel.setLayout(new BorderLayout()); //设置eastPanel布局为BorderLayout
37         eastPanel.add(mp, BorderLayout.NORTH); //将mp添加到eastPanel中
38         add(eastPanel, BorderLayout.EAST); //将eastPanel放入TestTransform
39         //添加Transform按钮并设置事件侦听器
40         Button button = new Button("Transform");
41         button.addActionListener(this);
42         Panel p = new Panel(); //创建Panel,用于放置Transform按钮,并置于eastPanel
43         p.add(button);
44         eastPanel.add(p, BorderLayout.EAST);
45         //生成新的面板Panel,设置布局为4行5列的网格
46         p = new Panel();
47         p.setLayout(new GridLayout(4, 5));
48         p.add(new Label("x")); //放置文本标签
49         p.add(new Label("y"));
50         p.add(new Label("z"));
51         p.add(new Label("angle"));
52         p.add(new Label(""));
```



```
53
54     p.add(rx); //放置Rotate相关文本字段
55     p.add(ry);
56     p.add(rz);
57     p.add(ra);
58     button = new Button("Rotate"); //添加Rotate按钮及事件侦听器
59     button.addActionListener(this);
60     p.add(button);
61
62     p.add(tx); //放置Translate相关文本字段
63     p.add(ty);
64     p.add(tz);
65     p.add(new Panel());
66     button = new Button("Translate"); //添加Translate按钮及事件侦听器
67     button.addActionListener(this);
68     p.add(button);
69
70     p.add(sx); //放置Scale相关文本字段
71     p.add(sy);
72     p.add(sz);
73     p.add(new Panel());
74     button = new Button("Scale"); //添加Scale按钮及事件侦听器
75     button.addActionListener(this);
76     p.add(button);
77
78     eastPanel.add(p, BorderLayout.SOUTH); //将p放入eastPanel
79
80     GraphicsConfiguration gc =
81         SimpleUniverse.getPreferredConfiguration();
82     Canvas3D cv = new Canvas3D(gc); //创建Canvas3D对象
83     add(cv, BorderLayout.CENTER);
84     BranchGroup bg = createSceneGraph();
85     bg.compile();
86     SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse对象
87     su.getViewingPlatform().setNominalViewingTransform();
88     su.addBranchGraph(bg);
89 }
90 //生成BranchGroup的私有方法
91 private BranchGroup createSceneGraph() {
92     BranchGroup root = new BranchGroup();
93     trGroup = new TransformGroup();
94     trGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
95     root.addChild(trGroup);
96
97     //创建形体
98     Appearance ap = new Appearance();
99     ap.setMaterial(new Material());
100     Group shape = new Axes();
101     //设置变换
102     Transform3D tr = new Transform3D();
103     tr.setScale(0.5);
104     TransformGroup tg = new TransformGroup(tr);
105     trGroup.addChild(tg);
106     tg.addChild(shape);
```

```

107
108 //设置背景和光照
109 BoundingSphere bounds = new BoundingSphere();
110 Background background = new Background(1.0f, 1.0f, 1.0f);
111 background.setApplicationBounds(bounds);
112 root.addChild(background);
113 AmbientLight light = new AmbientLight
220 (true, new Color3f(Color.red)); //设置环境光源
115 light.setInfluencingBounds(bounds);
116 root.addChild(light);
117 PointLight ptlight = new PointLight(new Color3f(Color.green),
118     new Point3f(3f, 3f, 3f), new Point3f(1f, 0f, 0f)); //设置点光源
119 ptlight.setInfluencingBounds(bounds);
120 root.addChild(ptlight);
121 PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
122     new Point3f(-2f, 2f, 2f), new Point3f(1f, 0f, 0f)); //设置点光源
123 ptlight2.setInfluencingBounds(bounds);
124 root.addChild(ptlight2);
125 return root;
126 }
127 //事件处理函数
128 public void actionPerformed(ActionEvent e) {
129     Matrix4d m = new Matrix4d();
130     mp.get(m); //从mp中的各文本字段获取数值并存入矩阵m
131     transform.set(m); //建立变换矩阵
132     String cmd = e.getActionCommand();
133     if ("Transform".equals(cmd)) {
134         Quat4d quat = new Quat4d();
135         Vector3d translation = new Vector3d();
136         transform.get(quat, translation);
137         Vector3d scale = new Vector3d();
138         transform.getScale(scale); //得到缩放组件
139         AxisAngle4d rotation = new AxisAngle4d();
140         rotation.set(quat); //将quat转化成AxisAngle4d类型
141         rx.setText("" + rotation.x);
142         ry.setText("" + rotation.y);
143         rz.setText("" + rotation.z);
144         ra.setText("" + rotation.angle);
145         tx.setText("" + translation.x);
146         ty.setText("" + translation.y);
147         tz.setText("" + translation.z);
148         sx.setText("" + scale.x);
149         sy.setText("" + scale.y);
150         sz.setText("" + scale.z);
151         trGroup.setTransform(transform);
152     } else {
153         if ("Rotate".equals(cmd)) { //旋转变换
154             double x = Double.parseDouble(rx.getText());
155             double y = Double.parseDouble(ry.getText());
156             double z = Double.parseDouble(rz.getText());
157             double a = Double.parseDouble(ra.getText());
158             transform.setRotation(new AxisAngle4d(x, y, z, a));
159         } else if ("Translate".equals(cmd)) { //平移变换
160             double x = Double.parseDouble(tx.getText());

```



```

161         double y = Double.parseDouble(ty.getText());
162         double z = Double.parseDouble(tz.getText());
163         transform.setTranslation(new Vector3d(x, y, z));
164     } else if ("Scale".equals(cmd)) { //缩放变换
165         double x = Double.parseDouble(sx.getText());
166         double y = Double.parseDouble(sy.getText());
167         double z = Double.parseDouble(sz.getText());
168         transform.setScale(new Vector3d(x, y, z));
169     }
170     transform.get(m);
171     mp.set(m);
172 }
173 }
174 }

```

221

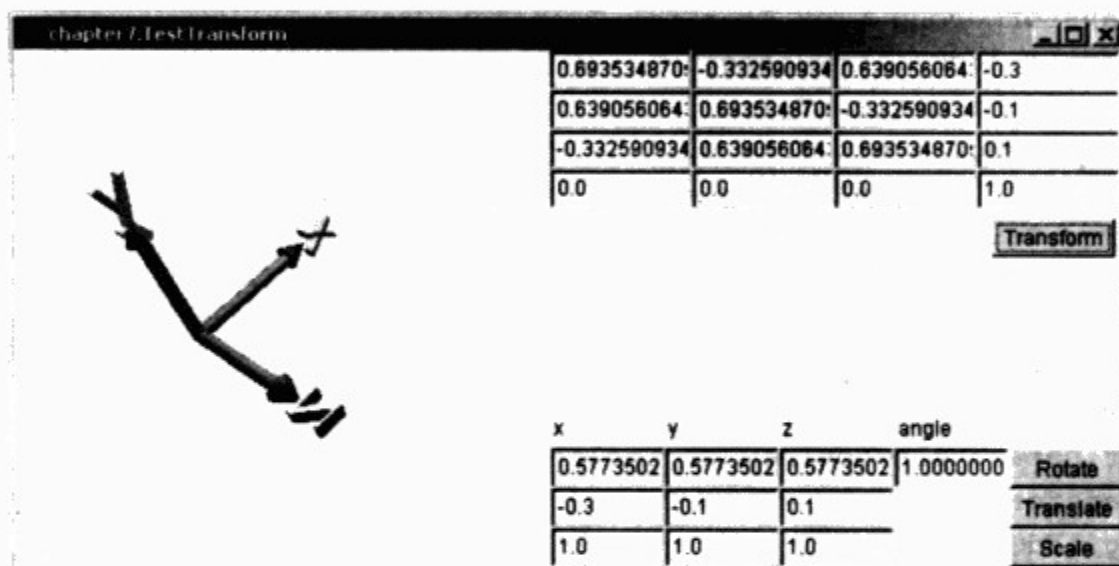


图7-2 可视化了变换的动作

程序清单7-5 Axes.java

```

1 package chapter7;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 //定义Axes类，继承自Group类，定义了3D坐标轴
10 public class Axes extends Group {
11     public Axes() {
12         Appearance ap = new Appearance(); //创建外观对象
13         ap.setMaterial(new Material());
14         Font3D font = new Font3D(new Font("SanSerif", Font.PLAIN, 1),
15                                 new FontExtrusion()); //创建3D字体对象
16         Text3D x = new Text3D(font, "x"); //创建3D文本对象
17         Shape3D xShape = new Shape3D(x, ap);
18         Text3D y = new Text3D(font, "y");
19         Shape3D yShape = new Shape3D(y, ap);
20         Text3D z = new Text3D(font, "z");
21         Shape3D zShape = new Shape3D(z, ap);

```

```

22    //文本的变换
23    Transform3D tTr = new Transform3D();
24    tTr.setTranslation(new Vector3d(-0.12, 0.6, -0.04));
25    tTr.setScale(0.5);
26    //箭头的变换
27    Transform3D aTr = new Transform3D();
28    aTr.setTranslation(new Vector3d(0, 0.5, 0));
29    //设置x轴
30    Cylinder xAxis = new Cylinder(0.05f, 1f);
31    Transform3D xTr = new Transform3D();
32    xTr.setRotation(new AxisAngle4d(0, 0, 1, -Math.PI/2));
33    xTr.setTranslation(new Vector3d(0.5, 0, 0));
34    TransformGroup xTg = new TransformGroup(xTr);
35    xTg.addChild(xAxis);
222 36    this.addChild(xTg);
37    TransformGroup xTextTg = new TransformGroup(tTr);
38    xTextTg.addChild(xShape);
39    xTg.addChild(xTextTg);
40    Cone xArrow = new Cone(0.1f, 0.2f);
41    TransformGroup xArrowTg = new TransformGroup(aTr);
42    xArrowTg.addChild(xArrow);
43    xTg.addChild(xArrowTg);
44    // 设置y轴
45    Cylinder yAxis = new Cylinder(0.05f, 1f);
46    Transform3D yTr = new Transform3D();
47    yTr.setTranslation(new Vector3d(0, 0.5, 0));
48    TransformGroup yTg = new TransformGroup(yTr);
49    yTg.addChild(yAxis);
50    this.addChild(yTg);
51    TransformGroup yTextTg = new TransformGroup(tTr);
52    yTextTg.addChild(yShape);
53    yTg.addChild(yTextTg);
54    Cone yArrow = new Cone(0.1f, 0.2f);
55    TransformGroup yArrowTg = new TransformGroup(aTr);
56    yArrowTg.addChild(yArrow);
57    yTg.addChild(yArrowTg);
58    //设置z轴
59    Cylinder zAxis = new Cylinder(0.05f, 1f);
60    Transform3D zTr = new Transform3D();
61    zTr.setRotation(new AxisAngle4d(1, 0, 0, Math.PI/2));
62    zTr.setTranslation(new Vector3d(0, 0, 0.5));
63    TransformGroup zTg = new TransformGroup(zTr);
64    zTg.addChild(zAxis);
65    this.addChild(zTg);
66    TransformGroup zTextTg = new TransformGroup(tTr);
67    zTextTg.addChild(zShape);
68    zTg.addChild(zTextTg);
69    Cone zArrow = new Cone(0.1f, 0.2f);
70    TransformGroup zArrowTg = new TransformGroup(aTr);
71    zArrowTg.addChild(zArrow);
72    zTg.addChild(zArrowTg);
73 }
74 }

```

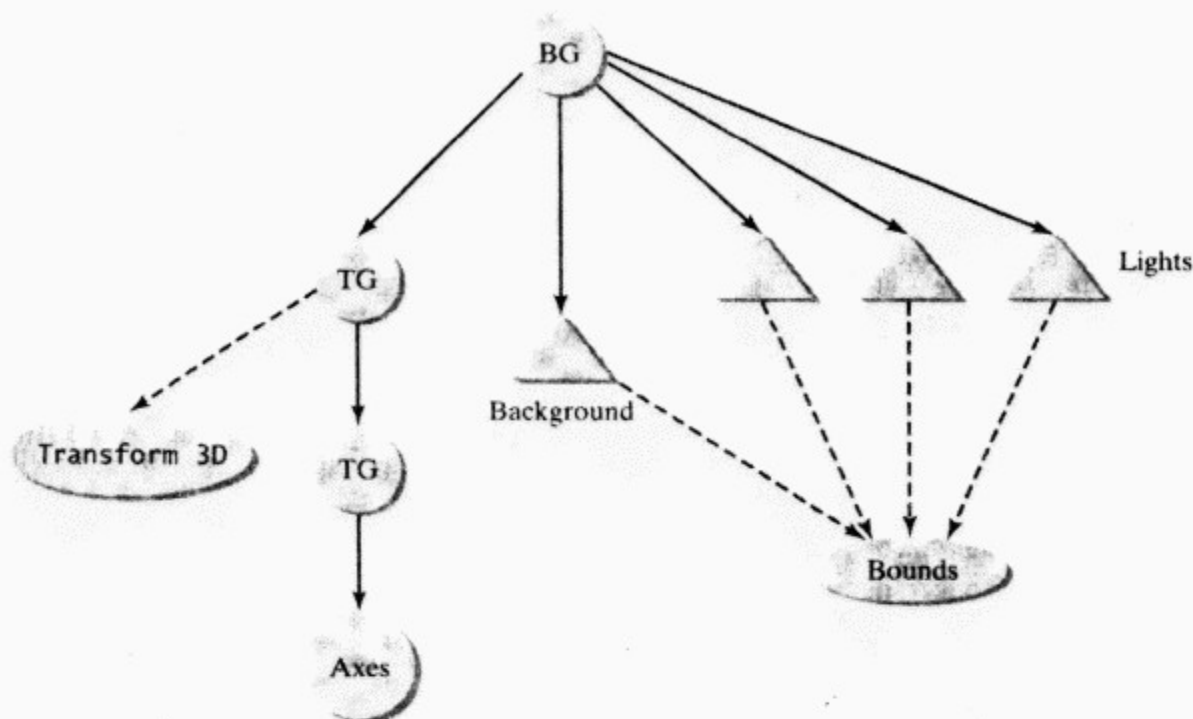



图7-3 TestTransform的场景图

223

TestTransform程序构造了一个窗口，以允许用户交互地修改变换并将变换应用于场景。在Canvas3D上绘制的Java 3D场景，是一组标记为x、y和z的3D坐标轴。这个可视对象是由Axes类定义的，而Axes类的具体讨论将在下一节给出。

Axes对象经过一个固定的TransformGroup节点来缩放其大小。形成的分支附属于另一个TransformGroup节点，这个节点允许动态地修改其变换。Transform3D对象控制这个节点，并且可以在运行时动态地进行设置。

类似于程序清单7-2，该程序生成了一个MatrixPanel对象来表示变换矩阵。旋转、平移和缩放等变换的不同内容经过提取以后，显示在几个TextField对象中。x、y和z部分也分别加以显示，旋转变换还包括一个旋转角度的定义。

当点击按钮“Transform”（第133行）时，将获取在MatrixPanel对象中指定的矩阵，根据该矩阵建立变换，从变换中重新计算旋转、平移和缩放。由四元数表示的旋转和由向量表示的平移，可以由下列Tranform3D方法得到：

```
double get(Quat4d rotation ,Vector3d translation)
```

上面的方法同样返回一个统一的缩放因子。要得到x、y和z方向上不同的缩放因子，可以使用下面的方法：

```
void getScale(Vector3d scale)
```

为了方便地得到旋转的轴和角度，通过get方法得到的Quat4d对象转化成了一个AxisAngle4d对象。提取出的旋转、平移和缩放变换显示于文本字段中。同时，用下面的方法，通过设置TransformGroup节点的变换，可以把指定的变换作用到3D场景上：

```
void setTranform(Transform3d transform)
```

标记为“Rotate”、“Translate”和“Scale”的按钮的动作，与标为“Transform”的按钮的动作相反。当点击“Rotate”按钮时，从文本字段中读出旋转的轴和角度，通过下面的方法用这些值对Transform3D对象进行旋转：

```
void setRotation(AxisAngle4d rotation)
```

类似地，“Translate”按钮和“Scale”按钮将指定的平移和缩放变换应用于变换。

7.3 场景图的变换

Java 3D场景图中的变换由TransformGroup类实现。TransformGroup对象定义了场景图的组节点，用于表示一个特定的变换，该组节点所引用的Transform3D对象定义了变换的具体形式。变换通常是仿射变换，但它也可以是Transform3D类所能表示的任何类型的变换。TransformGroup节点定义的变换将应用于它所有的子节点。

下面的代码显示了一个利用Matrix4d、Transform3D及TransformGroup对象建立变换节点的例子：

```
double[] array = {1.0 2.0 3.0 1.0,
                  0.0 1.0 -1.0 2.0,
                  4.0 0.0 0.5 -1.0,
                  0.0 0.0 0.0 1.0};

Matrix4d matrix = new Matrix4d(array);
Transform3D transform = new Transform3D(matrix);
TransformGroup node = new TransformGroup(transform);
```

224

TransformGroup节点可以用于改变3D世界空间中几何对象的形状、尺寸和位置。例如，上一章定义的Tetrahedron类是以原点为中心的，若要在场景图中放入一个Shape3D叶节点，它引用一个中心在(0.5, 0, -1)的Tetrahedron几何体，则需要生成一个平移(0.5, 0, -1)的TransformGroup节点，并将该四面体形状作为它的子节点：

```
Tetrahedron geom = new Tetrahedron();
Shape3D shape = new Shape3D();
Shape.setGeometry(geom);
Transform3D tr = new Transform3D();
tr.setTranslation(new Vector3d(0.5, 0, -1));
TransformGroup tg = new TransformGroup(tr);
tg.addChild(shape);
```

一个TransformGroup节点可以是另一个TransformGroup节点的子节点，可以构造多层的TransformGroup节点和其他节点，来表示几何模型的复杂结构。例如，一张桌子可以建模为四个桌腿和桌面。每个桌腿可以附属于一个TransformGroup节点，从而将桌腿移至相对于桌面的适当位置。如果想将整个桌子作为一个对象进行移动，必须在桌子模型的顶部再安放一个TransformGroup节点。

在程序清单7-5中，Axes类构造了三个相互垂直的坐标轴，每个轴包含一个圆柱体作为轴线，一个圆锥体作为箭头，以及一个3D文本作为标签。Axes的场景图见图7-4。

225

Cylinder类和Cone类允许在它们的构造函数中定义其半径和高。但是，它们的位置和方位是固定的，总是沿着y轴方向。要通过适当的变换来将它们移动到需要的位置，可以将y轴提高0.5使得它从原点出发。x轴和z轴除了平移变换之外，还需要旋转变换。轴上的箭头由Cone对象形成，它需要经过和对应的坐标轴一样的变换，并且还需要一个额外的平移变换，使箭头移动到轴的顶部。这个平移变换用一个单独的TransformGroup节点表示，它是作用于轴的变换节点的子节点，同时也是箭头节点的父节点。因此，这个平移变换只应用于圆锥体箭头，而轴的变换则同时应用于轴和平移后的箭头。类似地，文本标签“x”，“y”或“z”首先经过其自身的平移变换而到达相对于轴的适当位置，然后运用相同的轴变换，把标签、轴和箭头一起移动到最终的位置。

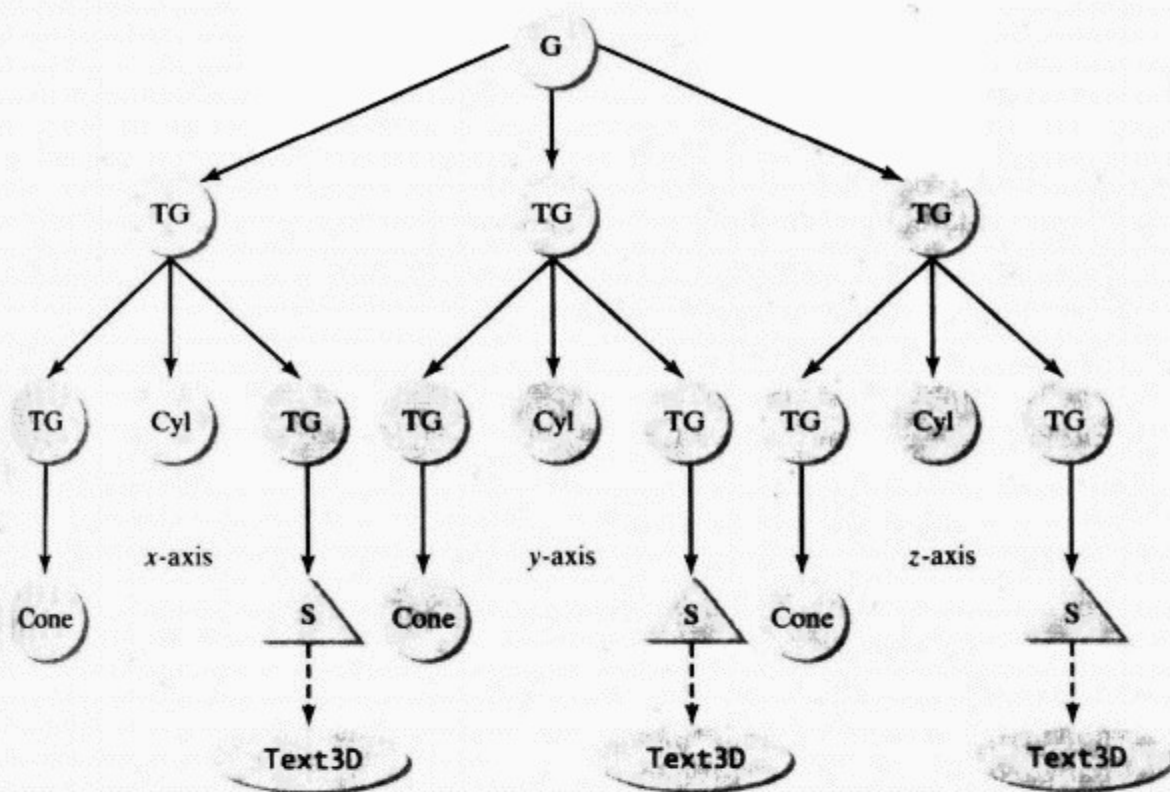


图7-4 Axes类的场景图

程序清单7-6 展示了TransformGroup节点的应用情况，以及关于任意轴的一般旋转的构造过程。程序通过显示处于旋转变换的不同阶段的八个立方体，来图示一个立方体绕任意轴的旋转变换（见图7-5）。八个立方体的不同位置由场景图中的各个TransformGroup节点获得。

程序清单7-6 Rotation.java

```

1 package chapter7;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义Rotation类，继承自Applet类
12 public class Rotation extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new Rotation(), 640, 480); //生成主窗口，设置窗口大小
15     }
16     //重写Applet的初始化函数
17     public void init() {
18         //生成canvas对象
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
21         Canvas3D cv = new Canvas3D(gc);
22         setLayout(new BorderLayout());
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph();
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse对象
27         su.getViewingPlatform().setNominalViewingTransform();
  
```



```

28     su.addBranchGraph(bg);
29 }
30 //生成BranchGroup的私有方法
31 private BranchGroup createSceneGraph() {
32     BranchGroup root = new BranchGroup();
33     Background background = new Background(1.0f, 1.0f, 1.0f);
34     BoundingSphere bounds = new BoundingSphere();
35     background.setApplicationBounds(bounds);
36     root.addChild(background);
37     Shape3D shape; //声明Shape3D形状对象
38     Appearance ap = new Appearance(); //创建外观对象
39     PolygonAttributes pa = new PolygonAttributes(); //创建多边形属性对象
40     ap.setPolygonAttributes(pa);
41     ColoringAttributes ca = new ColoringAttributes(0f, 0f, 0f,
42         ColoringAttributes.SHADE_FLAT); //创建并设置颜色属性对象
43     ap.setColoringAttributes(ca);
44     //旋转轴, 经过点(-0.8, -0.8, -0.8)和点(0.5, 0.5, 0.5)
45     LineArray axis = new LineArray(2, LineArray.COORDINATES);
46     axis.setCoordinate(0, new Point3d(-0.8, -0.8, -0.8));
47     axis.setCoordinate(1, new Point3d(0.5, 0.5, 0.5));
48
49     Shape3D axisG = new Shape3D(axis, ap);
50     root.addChild(axisG);
51     //平移变换和缩放变换
52     Transform3D tr = new Transform3D();
53     tr.setTranslation(new Vector3f(-0.5f, 0f, 0f));
54     tr.setScale(0.1);
55     TransformGroup tg;
56     TransformGroup rot;
57     for (int i = 0; i < 8; i++) { //生成不同阶段的八个立方体
58         shape = new ColorCube();
59         shape.setAppearance(ap);
60         tg = new TransformGroup(tr);
61         Transform3D trRot = new Transform3D(); //旋转变换
62         trRot.set(new AxisAngle4d(0.5, 0.5, 0.5, Math.PI/4*i));
63         rot = new TransformGroup(trRot);
64         root.addChild(rot);
65         rot.addChild(tg);
66         tg.addChild(shape);
67     }
68     return root;
69 }
70 }

```

程序的场景图见图7-6。程序生成了八个ColorCube对象, 它们将首先经过相同的平移和缩放变换, 来重新调整大小和位置, 然后每个变换过的立方体都附属于另一个TransformGroup点。将这些变换组节点设为围绕相同的轴旋转一系列取值均匀的不同角度, 这样就生成了一个显示绕轴旋转的各个不同阶段的立方体的场景。

旋转轴是一条经过原点和点(1, 1, 1)的直线。作为一个LineArray几何体, 这条直线也显示于场景中, 这条直线由两个点(-0.8, -0.8, -0.8)和(0.5, 0.5, 0.5)定义。

旋转由AxisAngle4d对象(第62行)生成。八个不同的旋转角度由下列表达式给出:

$$\frac{2\pi}{8}i, \quad i = 0, 1, \dots, 7$$

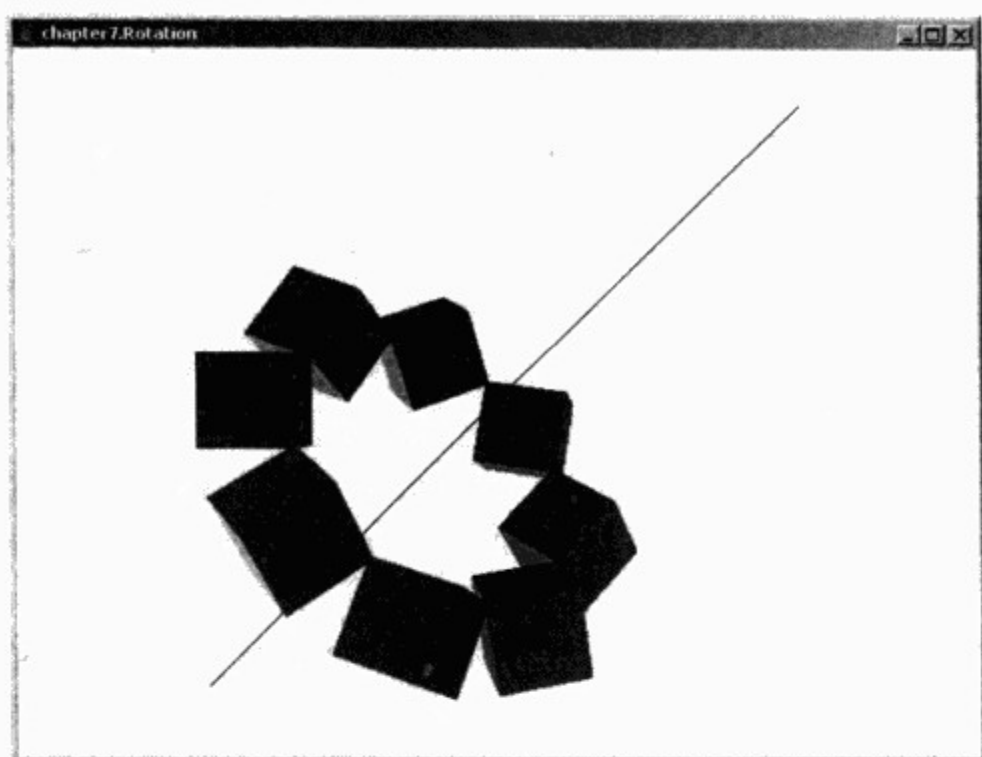


图7-5 绕一般轴的旋转的立方体

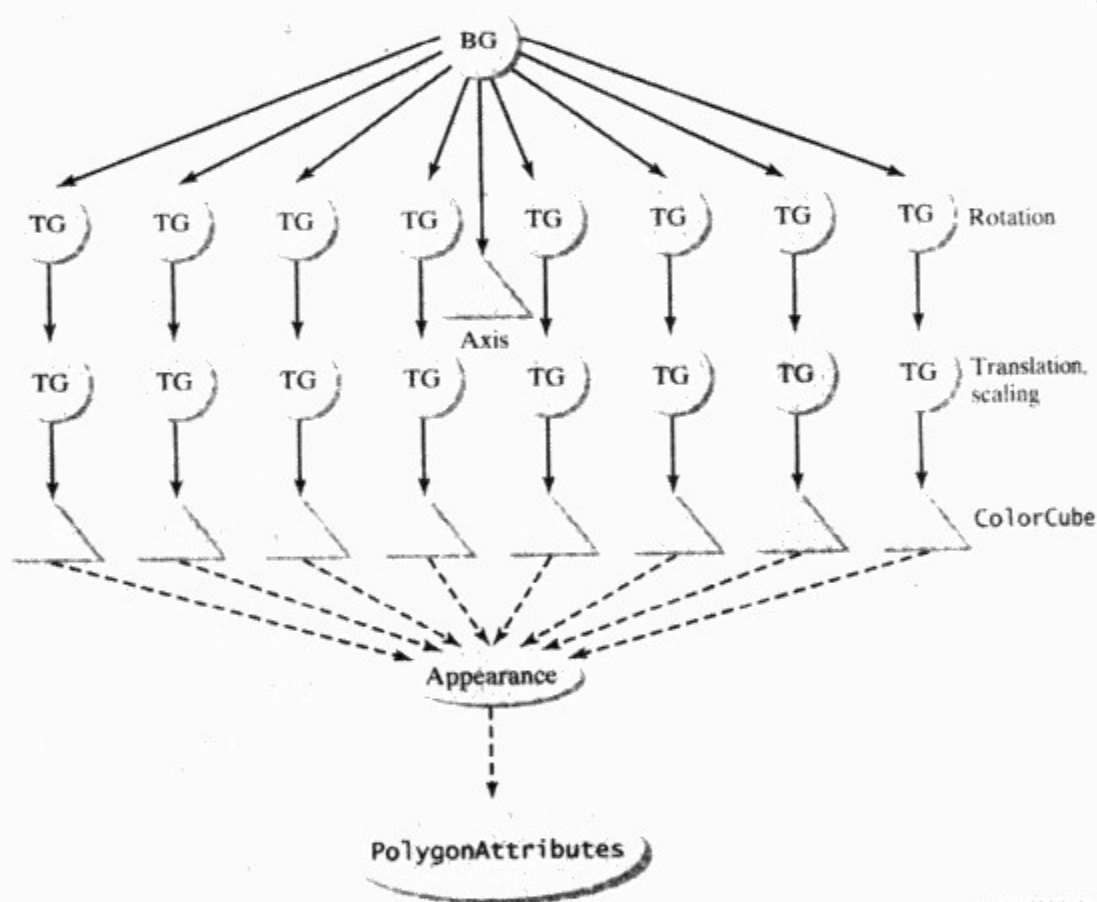


图7-6 Rotation 类的局部场景图

注意 为了表述清楚，旋转变换和最初的平移/缩放变换由不同层次的TransformGroup节点实现。如果将它们结合在一起，效率会更高。实际上，编译场景图时由Java 3D执行的优化操作，会自动把这种类型的变换节点结合起来。

7.4 复合变换

两个或更多的变换可以组合起来，形成一个复合变换。例如，如果 T_1 、 T_2 是两个变换，那么复合变换定义为：

$$(T_2 T_1)(p) = (T_2(T_1(p)))$$

注意到两个变换的复合不是可交换的, 一般来说 $T_2T_1 \neq T_1T_2$ 。在我们的表示方式中, 复合变换是从右至左进行结合的。例如, T_2T_1 表示先应用 T_1 , 然后应用 T_2 的复合变换。

在矩阵形式中, 变换的复合相当于矩阵的乘法。如果 T_1 、 T_2 的变换矩阵分别为 M_1 、 M_2 , 那么复合变换 T_2T_1 的变换矩阵为 M_2M_1 。

Transform3D 类包含许多方法, 用于将变换矩阵相乘, 以形成复合变换:

- **void mul(Transform3D t):** 当前变换右乘变换 t 。
- **void mul(Transform3D t1, Transform3D t2):** 变换 $t1$ 乘以变换 $t2$ 。
- **void mulInverse(Transform3D t):** 当前变换右乘以变换 t 的逆变换。
- **void mulInverse(Transform3D t1, Transform3D t2):** 变换 $t1$ 右乘以 $t2$ 的逆变换。
- **void mulTransposeBoth(Transform3D t1, Transform3D t2):** 变换 $t1$ 的转置右乘以变换 $t2$ 的转置。
- **void mulTransposeLeft(Transform3D t1, Transform3D t2):** 变换 $t1$ 的转置右乘以变换 $t2$ 。
- **void mulTransposeRight(Transform3D t1, Transform3D t2):** 变换 $t1$ 右乘以变换 $t2$ 的转置。

当要建立一个复杂的变换时, 比起直接构造其变换矩阵, 从一些较简单的变换组合得到该变换通常更为简单。

假设我们要构造一个旋转, 其旋转角度为 $\pi/3$, 旋转轴经过点 $(1,1,0)$ 和点 $(1,2,1)$ 。因旋转轴不经过原点, 不能直接应用四元数方法。但是, 我们可以首先做一个平移变换, 使得旋转轴经过原点, 然后再关于新的轴做旋转, 最后应用逆平移变换使得轴回到原来的位置。令 T 为平移变换 $(-1,-1,0)$, 那么 $T(1,1,0) = (0,0,0)$ 且 $T(1,2,1) = (0,1,1)$ 。令 R 是一个旋转变换, 旋转角为 $\pi/3$, 旋转轴经过 $(0,0,0)$ 和 $(0,1,1)$, 那么原旋转变换可以分解为 $T^{-1}RT$ 。

这种类型的分解很常见。如果已知一个特殊位置的特定变换, 那么在一般位置的类似变换可以这样得到: 首先通过变换移动到标准位置, 然后执行标准形式的给定变换, 最后通过变换回到原来的位置。

考虑关于反射变换的另一个例子。如先前所述, 关于 xy 平面的对称变换是很容易构造的。更一般的反射变换的变换矩阵较难求出, 但是, 我们可以将一般性的问题简化为较简单的关于 xy 平面的反射变换问题。假设对称平面是一个经过原点, 由等式 $ax + by + cz = 0$ 给出的平面, 可以利用复合变换而不是直接求变换矩阵。首先, 可以构造一个旋转变换, 将这个对称平面映射成 xy 平面, 然后执行关于 xy 平面的简单的反射变换, 最后用逆旋转变换将 xy 平面映射回原来的对称平面, 就此完成这个复合变换。这个复合变换和原来的反射变换是一致的。

为了得到将上述反射平面映射成 xy 平面的旋转变换, 相当于要将其法向量 (a,b,c) 映射成 xy 平面的法向量 $(0,0,d)$ 。所用变换的旋转轴为 $(b,-a,0)$ 。这个旋转变换的四元数有如下形式:

$$q = \cos(\theta/2) + \left(\frac{b}{\sqrt{a^2 + b^2}} i - \frac{a}{\sqrt{a^2 + b^2}} j \right) \sin(\theta/2)$$

旋转角度应该满足条件: $\cos \theta = c / \sqrt{a^2 + b^2 + c^2}$, $0 \leq \theta \leq \pi$,

将这个旋转变换记为 R , 而关于 xy 平面的反射变换记为 F 。那么, 关于这个平面的反射变换可以写成:

$$R^{-1}FR$$

程序清单7-7展示了利用这种方法的对称反射变换。位于一般位置的平面发挥着反射变换的“镜面”的作用。该程序将一个3D文本“Java”在场景中进行旋转, 它关于平面的镜像也进行了显示, 平面以半透明的形式进行显示, 以便显示通过反射变换得到的对象的镜面反射效果。

程序清单7-7 Mirror.Java

```
1 package chapter7;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义Mirror类,继承自Applet类,演示旋转物体的镜面图像效果
12 public class Mirror extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new Mirror(), 640, 480); //创建主窗口,设置窗口大小
15     }
16     //重写Applet初始化函数
17     public void init() {
18         //生成canvas
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
21         Canvas3D cv = new Canvas3D(gc);
22         setLayout(new BorderLayout());
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph();
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
27         su.getViewingPlatform().setNominalViewingTransform();
28         su.addBranchGraph(bg);
29     }
30
31     private BranchGroup createSceneGraph() { //生成场景图
32         //创建几何形体,为一个3D文本“Java”
33         Appearance ap = new Appearance();
34         ap.setMaterial(new Material());
35         Font3D font = new Font3D(new Font("Serif", Font.PLAIN, 1),
36             new FontExtrusion());
37         Shape3D shape = new Shape3D(new Text3D(font, "Java"), ap);
38         //平移
39         Transform3D trans = new Transform3D();
40         trans.setTranslation(new Vector3d(-0.5, 0, 0));
41         TransformGroup transg = new TransformGroup(trans);
42         transg.addChild(shape);
43         //旋转
44         TransformGroup spin = new TransformGroup();
45         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
46         spin.addChild(transg);
47         //缩放、平移
48         Transform3D tr = new Transform3D();
49         tr.setScale(0.25);
50         tr.setTranslation(new Vector3d(0.5, 0, 0));
51         TransformGroup tg = new TransformGroup(tr);
52         tg.addChild(spin);
```

```

53 //设置共享group
54 SharedGroup share = new SharedGroup();
55 share.addChild(tg);
56 //link leaf nodes to shared group
57 Link link1 = new Link(share);
58 Link link2 = new Link(share);
59 //对称
60 Transform3D reflection = getReflection(1,1,1);
61 TransformGroup reflectionGroup = new TransformGroup(reflection);
62 reflectionGroup.addChild(link2);
63 //对称平面 (镜面)
64 QuadArray qa = new QuadArray(4, QuadArray.COORDINATES);
65 qa.setCoordinate(0, new Point3d(0,-0.5,0.5));
66 qa.setCoordinate(1, new Point3d(1,-0.5,-0.5));
67 qa.setCoordinate(2, new Point3d(0,0.5,-0.5));
68 qa.setCoordinate(3, new Point3d(-1,0.5,0.5));
69 ap = new Appearance();
70 ap.setTransparencyAttributes(
71     new TransparencyAttributes
72     (TransparencyAttributes.BLENDED, 0.7f));
73 Shape3D mirror = new Shape3D(qa, ap);
74 //设置旋转器
75 Alpha alpha = new Alpha(-1, 4000);
76 RotationInterpolator rotator = new RotationInterpolator
77     (alpha, spin);
78 BoundingSphere bounds = new BoundingSphere();
79 rotator.setSchedulingBounds(bounds);
80 //设置背景和光照
81 Background background = new Background(0.5f, 0.5f, 0.5f);
82 background.setApplicationBounds(bounds);
83 AmbientLight light = new AmbientLight
84     (true, new Color3f(Color.red));
85 light.setInfluencingBounds(bounds);
86 PointLight ptlight = new PointLight(new Color3f(Color.green),
87     new Point3f(3f,3f,3f), new Point3f(1f,0f,0f));
88 ptlight.setInfluencingBounds(bounds);
89 PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
90     new Point3f(-2f,2f,2f), new Point3f(1f,0f,0f));
91 ptlight2.setInfluencingBounds(bounds);
92 //分支group
93 BranchGroup root = new BranchGroup();
94 root.addChild(link1);
95 root.addChild(reflectionGroup);
96 root.addChild(mirror);
97 root.addChild(rotator);
98 root.addChild(background);
99 root.addChild(light);
100 root.addChild(ptlight);
101 root.addChild(ptlight2);
102 return root;
103 }
104 //该方法计算关于给定平面的旋转变换
105 static Transform3D getReflection(double a, double b, double c) {
106     Transform3D transform = new Transform3D();
107     double theta = Math.acos(c/Math.sqrt(a*a+b*b+c*c));

```



```
108 double r = Math.sqrt(a*a+b*b);
109 Transform3D rot = new Transform3D();
110 rot.set(new AxisAngle4d(b/r, -a/r, 0, theta)); //设定旋转
111 Transform3D ref = new Transform3D();
112 ref.setScale(new Vector3d(1,1,-1)); //设定缩放
113 transform.mulInverse(rot); //根据上面的计算式计算复合变换
114 transform.mul(ref);
115 transform.mul(rot);
116 return transform;
117 }
118 }
```

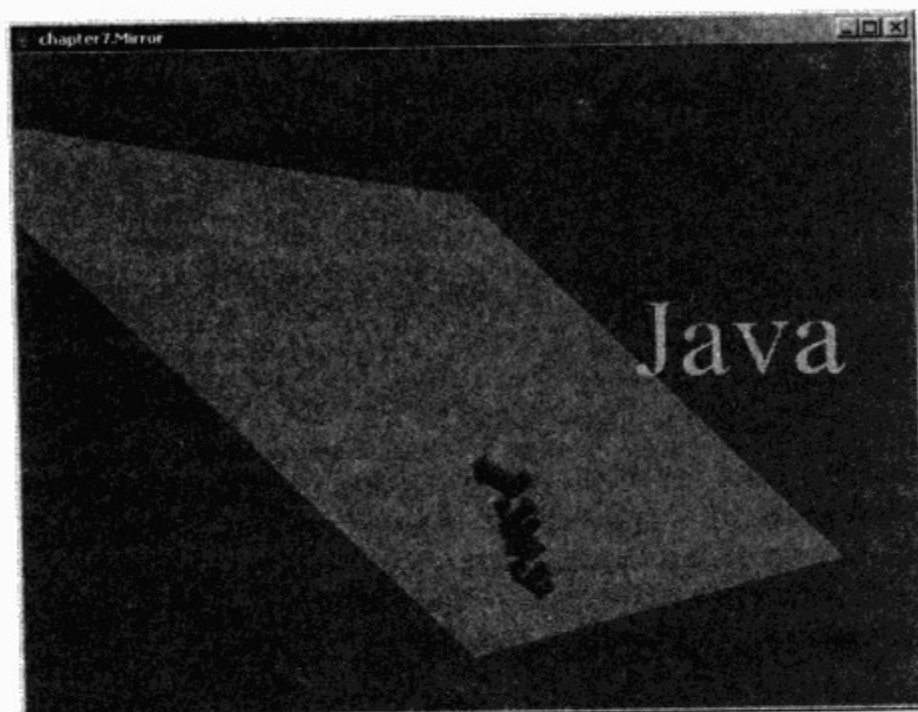


图7-7 由对称变换构造的镜像

程序构造了一面“镜子”，并且显示了一个旋转的对象及其镜像，其场景图如图7-8所示。

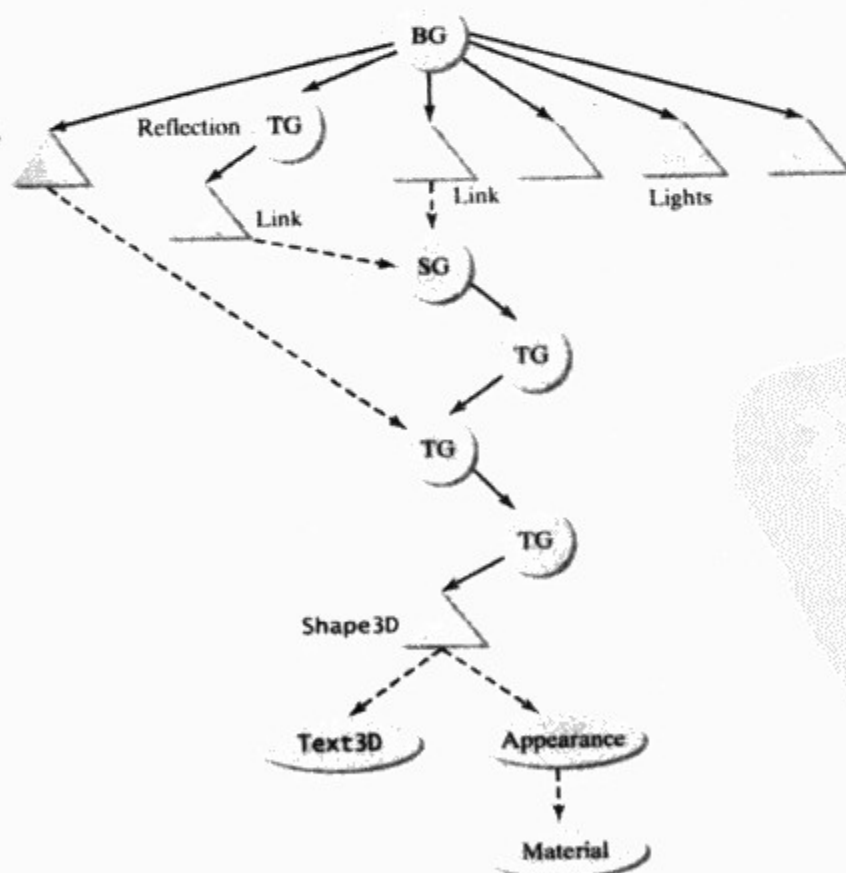


图7-8 Mirror类的场景图

镜子（即反射平面）选择为经过原点且法向量为（1,1,1）的平面，利用QuadArray对象（第64行），将镜子本身以半透明的矩形形式绘制于场景中。

getReflection方法（第105行）执行了上面关于给定平面的反射变换的推导，复合变换由一个旋转变换（将反射平面旋转成标准xy平面）、一个关于xy平面的简单反射变换以及一个逆旋转变换组成： $R^{-1}FR$ 。逆旋转变换可以通过mulInverse方法得到，对称与旋转由mul方法计算两者的乘，最终的复合变换作为一个Transform3D对象返回。

进行反射变换的对象是一个旋转的3D文本“Java”。为了忠实地显示对称变换，将这个可视对象放在一个SharedGroup分支之中，原对象和反射变换后的对象通过两个Link叶节点共享同一分支。在程序中，镜像对象的联系是通过反射对象来建立的。

7.5 用变换构造几何体

变换不仅在变换已完成构造的对象时有用，它还常用于构造几何基元。几何基元通常表现出某种程度的对称性，因此，几何基元的一些顶点可以通过其他顶点的变换得到。在这一节，我们将讨论利用仿射变换和其他操作生成3D几何体的几种技术。

Transform3D类提供了一些方法，用于将其表示的变换应用于点和向量：

```
void transform(Point3d p)
void transform(Point3d p, Point3d pOut)
void transform(Point3f p)
void transform(Point3f p, Point3f pOut)
void transform(Vector3d v)
void transform(Vector3d v, Vector3d vOut)
void transform(Vector3f v)
void transform(Vector3f v, Vector3f vOut)
void transform(Vector4d v)
void transform(Vector4d v, Vector4d vOut)
void transform(Vector4f v)
void transform(Vector4f v, Vector4f vOut)
```

带有一个参数的方法对参数所表示的点或向量进行变换，变换结果就保存在参数中。带有两个参数的方法对第一个参数进行变换，但不会改变第一个参数，而将变换结果保存在第二个参数中。

7.5.1 拉伸

在空间中拉伸（或扫描）2D曲线是一个生成3D曲面的简单方法，由Font3D类生成的3D文本就是拉伸的一个例子。

拉伸可以通过平移实现。从曲线上的点开始，我们沿着拉伸的方向应用一个平移变换来生成其他点。例如，下面的方法接受一个2DShape对象作为参数，并沿着z轴对它进行拉伸。

程序清单7-8 extrudeShape Method

```
1 //假定给定形体中只有一条连续曲线
2 Geometry extrudeShape(Shape curve, float depth) {
3     PathIterator iter = curve.getPathIterator(new AffineTransform());
4     Vector ptsList = new Vector();
5     float[] seg = new float[6];
6     float x = 0, y = 0;
7     float x0 = 0, y0 = 0;
8     while (!iter.isDone()) {
```



```

9      int segType = iter.currentSegment(seg);
10     switch (segType) {
11         case PathIterator.SEG_MOVETO:
12             x = x0 = seg[0];
13             y = y0 = seg[1];
14             ptsList.add(new Point3f(x,y,0));
15             break;
16         case PathIterator.SEG_LINETO:
17             x = seg[0];
18             y = seg[1];
19             ptsList.add(new Point3f(x,y,0));
20             break;
21         case PathIterator.SEG_QUADTO:
22             for (int i = 1; i < 10; i++) {
23                 float t = (float)i/10f;
24                 float xi = (1-t)*(1-t)*x + 2*t*(1-t)*seg[0] + t*t*seg[2];
25                 float yi = (1-t)*(1-t)*y + 2*t*(1-t)*seg[1] + t*t*seg[3];
26                 ptsList.add(new Point3f(xi,yi,0));
27             }
28             x = seg[2];
29             y = seg[3];
30             ptsList.add(new Point3f(x,y,0));
31             break;
32         case PathIterator.SEG_CUBICTO:
33             for (int i = 1; i < 20; i++) {
34                 float t = (float)i/20f;
35                 float xi = (1-t)*(1-t)*(1-t)*x + 3*t*(1-t)*(1-t)*seg[0] +
36                 3*t*t*(1-t)*seg[2] + t*t*t*seg[4];
37                 float yi = (1-t)*(1-t)*(1-t)*y + 3*t*(1-t)*(1-t)*seg[1] +
38                 3*t*t*(1-t)*seg[3] + t*t*t*seg[5];
39                 ptsList.add(new Point3f(xi,yi,0));
40             }
41             x = seg[2];
42             y = seg[3];
43             ptsList.add(new Point3f(x,y,0));
44             break;
45         case PathIterator.SEG_CLOSE:
46             x = x0;
47             y = y0;
48             ptsList.add(new Point3f(x,y,0));
49             break;
50     }
51     iter.next();
52 }
53 int n = ptsList.size();
54 IndexedQuadArray qa = new IndexedQuadArray(2*n,
55 IndexedQuadArray.COORDINATES, 4*(n-1));
56 Transform3D trans = new Transform3D();
57 trans.setTranslation(new Vector3f(0,0,depth));
58 for (int i = 0; i < n; i++) {
59     Point3f pt = (Point3f)ptsList.get(i);
60     qa.setCoordinate(2*i, pt);
61     trans.transform(pt);
62     qa.setCoordinate(2*i+1, pt);

```



```

63     }
64     int quadIndex = 0;
65     for (int i = 0; i < n-1; i++) {
66         qa.setCoordinateIndex(quadIndex++, 2*i);
67         qa.setCoordinateIndex(quadIndex++, 2*i+1);
68         qa.setCoordinateIndex(quadIndex++, 2*(i+1)+1);
69         qa.setCoordinateIndex(quadIndex++, 2*(i+1));
70     }
71     GeometryInfo gi = new GeometryInfo(qa);
72     NormalGenerator ng = new NormalGenerator();
73     ng.generateNormals(gi);
74     return gi.getGeometryArray();
75 }

```

为简单起见，这个方法假设Shape对象只定义了一条连续曲线，一般的情况可以用类似的方式处理。参数depth定义了拉伸的深度。

7.5.2 旋转

许多曲面都可以通过旋转得到。例如，旋转一条直线可以得到一个柱面，而一个球体是旋转一个半圆的结果。关于圆外的轴对圆进行旋转，可以构造一个圆环面（torus）。而且，圆本身可以通过旋转一个点得到。因此，我们可以从一个单独的点出发，得到一个圆环面的多边形网格，以不同的角度旋转点可以生成圆的点集，围绕另一个不同的轴旋转这些圆上的点，可以生成圆环面的顶点。图7-9描述了这个过程。

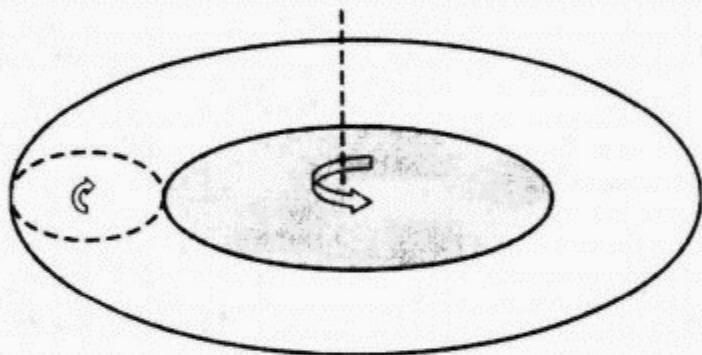


图7-9 通过旋转生成圆环面

给出曲线上的一组基点，要生成 n 个带组成的旋转曲面，需要建立一个旋转角为 $2\pi/n$ 的旋转变换。通过依次将旋转变换应用于曲线上的点，我们可以为一个四边形数组生成顶点。程序清单7-9通过

两个不同的旋转变换构造一个圆环面，程序清单7-10显示两个圆环面，圆环面几何体的顶点通过应用变换生成。在这个测试程序中，显示了经过空间旋转而得到的两个互相缠绕的圆环面。

程序清单7-9 Torus.java

```

1 package chapter7;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 //定义Torus类，继承自Shape3D类，用于定义圆环体
10 public class Torus extends Shape3D {
11     public Torus(double r1, double r2) {
12         int m = 20;
13         int n = 40;
14         Point3d[] pts = new Point3d[m];
15         pts[0] = new Point3d(r1+r2, 0, 0);
16         double theta = 2.0 * Math.PI / m;

```

//用于存储生成的圆上的点的数组


```

17     double c = Math.cos(theta);
18     double s = Math.sin(theta);
19     double[] mat = {c, -s, 0, r2*(1-c),      //旋转rot1的变换矩阵
20                     s, c, 0, -r2*s,
21                     0, 0, 1, 0,
22                     0, 0, 0, 1};
23     Transform3D rot1 = new Transform3D(mat); //从一个点生成圆上点的旋转
24     for (int i = 1; i < m; i++) {           //生成圆上的点, 储存于pts中
25         pts[i] = new Point3d();
26         rot1.transform(pts[i-1], pts[i]);
27     }
28
29     Transform3D rot2 = new Transform3D();   //从圆上点生成圆环面的旋转
30     rot2.rotY(2.0*Math.PI/n);              //将旋转设为绕y轴, 角度为2π/n
31     IndexedQuadArray qa = new IndexedQuadArray(m*n,
32         IndexedQuadArray.COORDINATES, 4*m*n); //用于构造几何体
33     int quadIndex = 0;                     //索引
34     for (int i = 0; i < n; i++) {
35         qa.setCoordinates(i*m, pts);       //设定顶点坐标数组
36         for (int j = 0; j < m; j++) {
37             rot2.transform(pts[j]);        //旋转圆上的点
38             int[] quadCoords = {i*m+j, ((i+1)%n)*m+j,
39                                 ((i+1)%n)*m+((j+1)%m), i*m+((j+1)%m)};
40             qa.setCoordinateIndices(quadIndex, quadCoords); //设定顶点坐标索引
41             quadIndex += 4;
42         }
43     }
44     GeometryInfo gi = new GeometryInfo(qa);
45     NormalGenerator ng = new NormalGenerator(); //生成法向量
46     ng.generateNormals(gi);
47     this.setGeometry(gi.getGeometryArray());
48 }
49 }
50

```

236

程序清单7-10 TestTorus.java

```

1 package chapter7;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义TestTorus类, 继承自Applet, 用于测试Torus类
12 public class TestTorus extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new TestTorus(), 640, 480); //创建主窗口, 设定窗口大小
15     }
16     //重写Applet初始化函数
17     public void init() {

```

```

18    //生成canvas
19    GraphicsConfiguration gc =
20    SimpleUniverse.getPreferredConfiguration();
21    Canvas3D cv = new Canvas3D(gc);
22    setLayout(new BorderLayout());
23    add(cv, BorderLayout.CENTER);
24    BranchGroup bg = createSceneGraph();
25    bg.compile();
26    SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
27    su.getViewingPlatform().setNominalViewingTransform();
28    su.addBranchGraph(bg);
29 }
30 //生成BranchGroup的私有方法
31 private BranchGroup createSceneGraph() {
32     BranchGroup root = new BranchGroup();
33     TransformGroup spin = new TransformGroup();
34     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
35     root.addChild(spin);
36     //设置旋转变换
37     Transform3D tr = new Transform3D();
38     tr.setScale(0.8);
39     tr.setRotation(new AxisAngle4d(1, 0, 0, Math.PI/6));
40     TransformGroup tg = new TransformGroup(tr);
41     spin.addChild(tg);
42
43     //创建形体
44     Shape3D torus1 = new Torus(0.2, 0.5); //第一个圆环体torus1
45     Appearance ap = new Appearance();
46     ap.setMaterial(new Material());
47     torus1.setAppearance(ap);
48     tg.addChild(torus1);
49
50     Shape3D torus2 = new Torus(0.2, 0.5); //第二个圆环体torus2
51     ap = new Appearance();
52     ap.setMaterial(new Material());
53     ap.setTransparencyAttributes(
54         new TransparencyAttributes
55         (TransparencyAttributes.BLENDED, 0.5f)); //设为半透明的
56     torus2.setAppearance(ap);
57     Transform3D tr2 = new Transform3D();
58     tr2.setRotation(new AxisAngle4d(1, 0, 0, Math.PI/2));
59     tr2.setTranslation(new Vector3d(0.5, 0, 0));
60     TransformGroup tg2 = new TransformGroup(tr2);
61     tg.addChild(tg2);
62     tg2.addChild(torus2);
63     //设置旋转
64     Alpha alpha = new Alpha(-1, 8000);
65     RotationInterpolator rotator = new RotationInterpolator
66         (alpha, spin);
67     BoundingSphere bounds = new BoundingSphere();
68     rotator.setSchedulingBounds(bounds);
69     spin.addChild(rotator);
70
71     //设置背景和光照

```



```
72     Background background = new Background(1.0f, 1.0f, 1.0f);
73     background.setApplicationBounds(bounds);
74     root.addChild(background);
75     AmbientLight light = new AmbientLight
76         (true, new Color3f(Color.blue)); //环境光源
77     light.setInfluencingBounds(bounds);
78     root.addChild(light);
79     PointLight ptlight = new PointLight(new Color3f(Color.white),
80         new Point3f(3f,3f,3f), new Point3f(1f,0f,0f)); //点光源
81     ptlight.setInfluencingBounds(bounds);
82     root.addChild(ptlight);
83     return root;
84 }
85 }
```

Torus类继承自Shape3D类，该几何体由IndexedQuadArray类构造。旋转变换rot1（第23行）从一个点生成一个圆上的点，生成的圆上的点存储于数组pts中。另一个旋转变换rot2（第29行）用于从圆上的点生成圆环面上的点，每次生成一系列圆环面上的网格点，且这些点存入IndexedQuadArray中（第35行），然后设置了各四边形的顶点索引（第40行）。为了生成法线，将IndexedQuadArray对象转化为GeometryInfo对象，并用NormalGenerator来自动生成几何体的曲面法线。

TestTorus类显示了两个不同位置的Torus类实例（见图7-10），它们的位置使得其中一个圆环面穿过另一个的洞眼。利用一个TransparencyAttributes对象将其中一个圆环面设成半透明的，整个图形一直在旋转，以便于从不同角度进行观察。

238

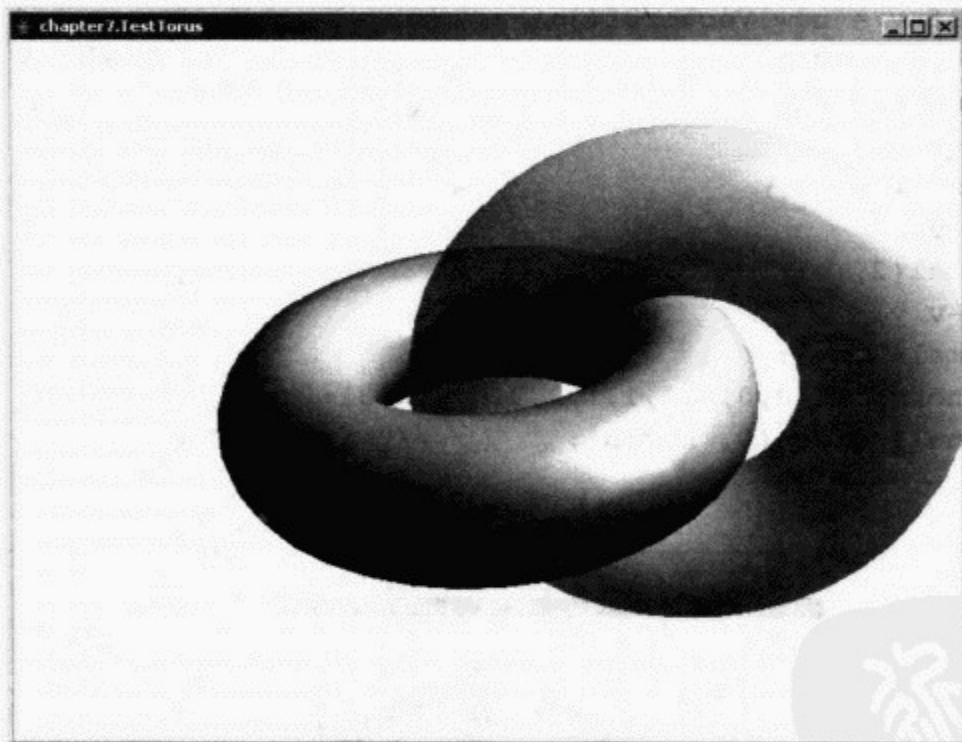


图7-10 两个Torus几何基元的实例，它们的顶点通过两组旋转变换生成

7.5.3 变换和共享分支的实例

复杂的可视内容的构造，通常要应用不同层次的变换。程序清单7-11给出了一个显示箭头几何体的类，程序清单7-12描述了如何通过共享分支和变换的组合，达到重用对称子结构的目的。程序显示了一个由16个箭头和一个圆环构成的3D图标，它围绕一根竖直轴旋转（见图7-11）。

程序清单7-11 Arrow.java

```

1 package chapter7;
2
3 import javax.vecmath.*;
4 import javax.media.j3d.*;
5 //定义Arrow类, 继承自IndexedTriangleArray类, 用于生成箭头几何体
6 public class Arrow extends IndexedTriangleArray {
7     float w = 1f;
8     float h = 0.15f;
9     float d = 0.1f;
10
11     public Arrow() {
12         super(5, TriangleArray.COORDINATES | TriangleArray.NORMALS, 12);
13         Point3f[] pts = {new Point3f(0f,0f,d), //定义点坐标数组
14             new Point3f(w,0f,0f),
15             new Point3f(h,h,0f),
16             new Point3f(h,-h,0f),
17             new Point3f(0f,0f,-d)};
18         setCoordinates(0, pts); //设定点坐标数组
19         int[] coords = {0,1,2,0,3,1,4,1,3,4,2,1};
20         setCoordinateIndices(0, coords); //设定顶点坐标索引数组
21         Vector3f v1 = new Vector3f();
22         v1.sub(pts[1], pts[0]); //v1等于pts[1]减去pts[0]
23         v1.normalize(); //将v1归一化
24         Vector3f v2 = new Vector3f();
25         v2.sub(pts[2], pts[0]); //v2等于pts[2]减去pts[0]
26         v2.normalize(); //将v2归一化
27         Vector3f v = new Vector3f();
28         v.cross(v1, v2); //计算向量外积, 生成并设置法向量
29         setNormal(0, v);
30         v.y = -v.y;
31         setNormal(1, v);
32         v.z = -v.z;
33         setNormal(2, v);
34         v.y = -v.y;
35         setNormal(3, v);
36         int[] norms = {0,0,0,1,1,1,2,2,2,3,3,3};
37         setNormalIndices(0, norms); //设置法向量索引
38     }
39 }

```

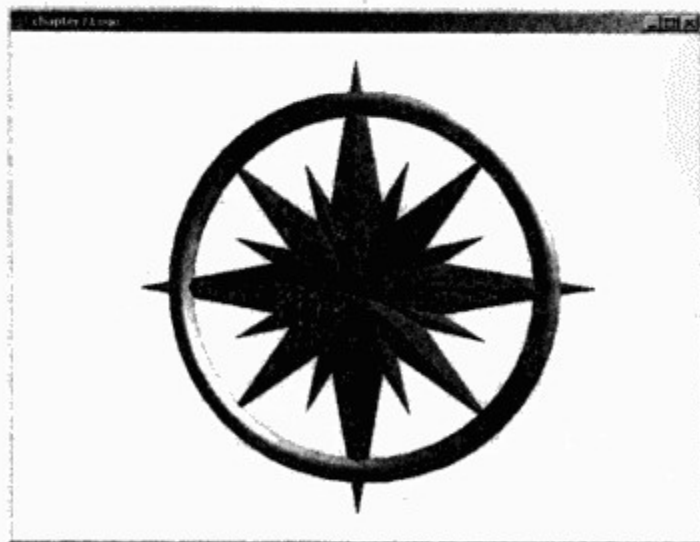


图7-11 一个3D标志

程序清单7-12 Logo.java

```
1 package chapter7;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11
12 public class Logo extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new Logo(), 640, 480);
15     }
16
17     public void init() { //初始化
18         //生成画布
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
21         Canvas3D cv = new Canvas3D(gc);
22         setLayout(new BorderLayout());
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph();
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv);
27         su.getViewingPlatform().setNominalViewingTransform();
28         su.addBranchGraph(bg);
29     }
30
31     private BranchGroup createSceneGraph() { //生成场景图
32         BranchGroup root = new BranchGroup();
33         TransformGroup spin = new TransformGroup();
34         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
35         root.addChild(spin);
36
37         Transform3D tr = new Transform3D();
38         tr.setScale(0.9);
39         tr.setRotation(new AxisAngle4d(1, 0, 0, Math.PI/2));
40         TransformGroup tg = new TransformGroup(tr);
41         spin.addChild(tg);
42         // 圆环面 (即logo中的圆环)
43         Shape3D torus = new Torus(0.04, 0.6);
44         Appearance ap = new Appearance();
45         ap.setMaterial(new Material());
46         torus.setAppearance(ap);
47         tg.addChild(torus);
48         //四个箭头的共享组
49         SharedGroup sg = new SharedGroup();
50         Shape3D arrow;
51         Transform3D tra;
52         TransformGroup tga;
53         for (int i = 0; i < 4; i++) {
```

240

```
54     arrow = new Shape3D(new Arrow(), ap);
55     tra = new Transform3D();
56     tra.setRotation(new AxisAngle4d(0, 0, 1, i*Math.PI/2));
57     tga = new TransformGroup(tra);
58     sg.addChild(tga);
59     tga.addChild(arrow);
60 }
61 //链接到共享组的四个link
62 Link link = new Link();
63 link.setSharedGroup(sg);
64 tr = new Transform3D();
65 tr.setScale(0.675);
66 tg = new TransformGroup(tr);
67 tg.addChild(link);
68 spin.addChild(tg);
69
70 link = new Link();
71 link.setSharedGroup(sg);
72 tr = new Transform3D();
73 tr.setScale(0.55);
74 tr.setRotation(new AxisAngle4d(0, 0, 1, Math.PI/4));
75 tg = new TransformGroup(tr);
76 tg.addChild(link);
77 spin.addChild(tg);
78
79 link = new Link();
80 link.setSharedGroup(sg);
81 tr = new Transform3D();
82 tr.setScale(0.4);
83 tr.setRotation(new AxisAngle4d(0, 0, 1, Math.PI/8));
84 tg = new TransformGroup(tr);
85 tg.addChild(link);
86 spin.addChild(tg);
87
88 link = new Link();
89 link.setSharedGroup(sg);
90 tr = new Transform3D();
91 tr.setScale(0.4);
92 tr.setRotation(new AxisAngle4d(0, 0, 1, 3*Math.PI/8));
93 tg = new TransformGroup(tr);
94 tg.addChild(link);
95 spin.addChild(tg);
96 //旋转变换
97 Alpha alpha = new Alpha(-1, 8000);
98 RotationInterpolator rotator =
99     new RotationInterpolator(alpha, spin);
100 BoundingSphere bounds = new BoundingSphere();
101 rotator.setSchedulingBounds(bounds);
102 spin.addChild(rotator);
103 //背景和光照
104 Background background = new Background(1.0f, 1.0f, 1.0f);
105 background.setApplicationBounds(bounds);
106 root.addChild(background);
107 AmbientLight light = new AmbientLight
```



```

108     (true, new Color3f(Color.red));
109     light.setInfluencingBounds(bounds);
110     root.addChild(light);
111     PointLight ptlight = new PointLight(new Color3f(Color.white),
112     new Point3f(2f,2f,2f), new Point3f(1f,0f,0f));
113     ptlight.setInfluencingBounds(bounds);
114     root.addChild(ptlight);
115     return root;
116 }
117 }

```

圆环是程序清单7-6中定义的Torus类的一个实例，16个箭头具有高度的对称性和相似性。为了利用好这些对称性以简化建模过程，我们应用了变换和共享分支。

Arrow类定义了单个箭头图元的几何属性，它是IndexTriangleArray的派生类。这个几何体定义了五个顶点和四个面，一个面的曲面法向量由两条边向量的叉积计算得到。利用对称性，其他面的法线可以由第一个面的法线得到。

242

通过变换，四个Arrow对象组合成一组，形成一个共享分支（第48~60行），这四个箭头的方向分别为0°、90°、180°、270°。

对应全部的16个箭头，程序生成了指向SharedGroup节点的四个不同的Link对象（第61~95行），每个Link对象都经过了适当角度的旋转变换和缩放变换。

类似其他许多例子，利用一个RotationInterpolator对象，使组合而成的整个图标在场景中做旋转运动，这个图标有一个包含了Material对象的外套环，它受到两个光源的光照，场景的背景还是设为白色。

主要的类及方法

- **javax.vecmath.Matrix4d** 封装了 4×4 double类型矩阵的类。
- **javax.vecmath.Matrix4f** 封装了 4×4 float矩阵类型的类。
- **javax.media.j3d.Transform3D** 一个封装了3D变换的类。
- **javax.media.j3d.Transform3D.set(...)** 设置变换的方法。
- **javax.media.j3d.Transform3D.setTranslation(...)** 建立变换的平移部分的方法。
- **javax.media.j3d.Transform3D.setRoation(...)** 建立变换的旋转部分的方法。
- **javax.media.j3d.Transform3D.setScale(...)** 建立变换的缩放部分的方法。
- **javax.media.j3d.Transform3D.mul(...)** 乘以另一个变换的方法。
- **javax.media.j3d.Transform3D.mulInverse(...)** 乘以另一个变换的逆变换的方法。
- **javax.media.j3d.Transform3D.transform(...)** 将变换应用于点和向量的方法。
- **javax.media.j3d.TransformGroup** 变换组节点类。
- **javax.vecmath.Quat4d** 封装了double类型的四元数的类。
- **javax.vecmath.Quat4f** 封装了float类型的四元数的类。

关键术语

- **仿射变换** (affine transform) 保持平行性的几何变换。
- **投影变换** (projective transform) 保持点、直线及其入射关系的几何变换。
- **欧拉角** (Euler angles) 用绕三个坐标主轴的旋转来定义3D旋转变换的方法。
- **复合变换** (composite transformation) 由若干个变换的积定义的变换。
- **3D变换矩阵** (3D transformation matrix) 一个表示3D投影变换的矩阵。

- **四元数表示法** (quaternion representation) 用四元数来表示3D旋转的方法。
- **3D平移变换** (3D translation) 将所有点移动一个常量的几何变换。
- **3D旋转变换** (3D rotation) 将点围绕一个固定轴旋转一定角度的几何变换。
- **3D缩放变换** (3D scaling) 将坐标缩放一定因子的几何变换。当x、y和z方向的三个因子相等时，缩放是均匀的。
- **3D错切变换** (3D shearing) 将点平行于平面移动的几何变换。
- **3D反射变换** (3D reflection) 将点移动到关于某个固定平面的对称位置的几何变换。

243

本章提要

- 这一章中，我们介绍了3D变换，尤其是仿射变换族的概念和应用。
- 3D变换用矩阵表示，矩阵类由包javax.vecmath提供。在Java 3D中，Transform3D类封装了变换矩阵和与变换相关的各种运算，这个类包含了大量方法，用于设置变换、执行矩阵运算、组合其他变换，以及对点和向量应用变换等。
- TransformGroup节点代表了Java场景图中的变换。一个TransformGroup节点通过引用一个Transform3D对象来定义它的变换，并将变换应用于它的子节点。
- 3D旋转是复杂的变换。本章介绍了旋转的三种不同的表示方法，矩阵表示法和其他仿射变换一致，但是它和几何参数的关系通常是隐式的。由于和旋转轴和旋转角度的直接联系，四元数表示法可以显式地表示一般的3D旋转，使用起来十分方便。欧拉角则提供了另一种直观的3D旋转的表示方法。
- 变换的复合是很有用的，尤其是在构造复杂的变换时。一个复杂的变换通常可以分解为一些较简单的变换的复合变换。
- 变换的另一个应用是几何体的构造。几何图像的对称性和规则性通常与一些变换相关联。位于这类几何体上的点，可以方便地通过变换生成。扫过一条曲线可以得到一个拉伸的曲面，平移和拉伸相关，旋转曲面通过曲线的旋转生成。

复习题

- 7.1 求出将点 (3, -1, 2) 映射成 (0, 5, -1) 的平移变换的变换矩阵。
- 7.2 求出绕y轴、角度为30度的旋转变换的变换矩阵。
- 7.3 求出关于经过原点且法向量为 (1, 1, 1) 的平面的反射变换的变换矩阵。
- 7.4 设一个反射变换由下式定义

$$T(x) = x - \frac{2x \cdot u}{\|u\|^2} u$$

其对称平面经过原点且法向量为u，试推导其变换矩阵。

244

- 7.5 计算四元数的积：(1 + 3i - j + k) · (2i + j - 3k)。
- 7.6 令 $u = (x_u, y_u, z_u)$ 和 $v = (x_v, y_v, z_v)$ 是两个正交单位向量，求出将u映成x轴、v映成y轴以及 $u \times v$ 映成z轴的旋转。
- 7.7 一个 (x, y) 错切变换由以下矩阵定义：

$$\begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

若给出正交单位向量u和v，试推导一复合变换，该变换执行因子为 (sh_u, sh_v) 的 (u, v) 错切变换。

- 7.8 求出由四元数 $q = w + xi + yj + zk$ 表示的3D旋转变换的变换矩阵。
- 7.9 如果R_x是一个围绕x轴的角度为30°的旋转，而R_y是一个围绕y轴的角度为60°的旋转，试根据旋转轴和

角度来描述复合变换 $R_y R_x$ 。

编程练习

- 7.1 写一个Java 3D程序，用于显示两个有公共三角形面的四面体。
- 7.2 显示两个顶点互相指向对方的圆锥体。
- 7.3 显示一个顶点指向方向(1,1,1)的圆锥体。
- 7.4 修改程序清单7-4，使其包含一个(x,y)错切变换，允许用户指定变换因子(sh_x, sh_y)。
- 7.5 修改程序清单7-4，使其包含由欧拉角描述的旋转。
- 7.6 修改程序清单7-7，直接从下列公式构造反射变换的矩阵

$$T(x) = x - \frac{2x \cdot u}{\|u\|^2} u$$

- 7.7 生成并显示一张简易的桌子，它有一个正方形的桌面及四个圆柱桌腿。只利用预定义几何基元及变换来生成这个可视对象，在场景中连续地旋转这个桌子。
- 7.8 生成这样一个几何体，它由图7-12中的图形绕y轴旋转而形成，写一个测试程序来显示这个对象。
- 7.9 实现一个Shape3D类RotatedShape，表示通过绕y轴旋转一个2DShape对象而得到的曲面。它至少要有一个构造函数：

```
public RotatedShape(Shape curve)
```

可以假设这个2D形体是一条连续曲线（只有一个SEG_MOVETO），写一个测试程序，显示这个曲面的动态旋转效果。

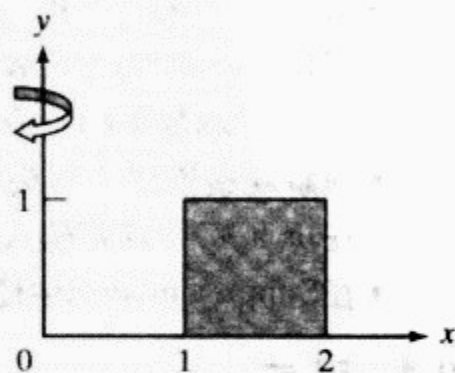


图7-12 旋转一个正方形来生成一个3D对象

245

246

第8章 视图

学习目标

- 描述3D绘制过程的视图概念。
- 认识平行投影与透视投影。
- 定义观察矩阵。
- 定义投影矩阵。
- 运用Java 3D标准视图模型。
- 运用Java 3D兼容模式视图模型。
- 运用3D场景中的拾取算法。
- 理解观察模型中的头部跟踪。
- 在Java 3D中使用输入设备、传感器和头部跟踪。
- 在SimpleUniverse中使用替身。

247

8.1 引言

当构建完虚拟世界中的图形对象之后，就可以通过观察变换过程，生成虚拟世界的各种视角的图像。场景绘制过程受多种属性影响，这些属性影响了视图的不同方面。本章将着重介绍绘制过程的几何方面，而获得绘制可见物体的真实感外观的方法，将在后续的章节中讨论。

定义从3D虚拟世界场景到2D图像映射过程的几何配置，称为视图（view）。视图是照相机的数字模拟形式，它定义了虚拟世界和图形对象实现可视化的方式。现代计算机图形系统的视图非常复杂，可能需要大量的参数。通常来讲，视图的视点位于虚拟世界中，从特定的观察方向和方位（orientation and direction）“观看”虚拟世界。每个视图具有某些定义属性，比如3D到2D的投影类型、视域、前后端裁剪平面、视图平面大小等。可以说，视图模型有点像静态照相机，还可以说，视图的工作机制更像人的眼睛，它可以动态地改变自己的观察位置与属性。

大多数底层图形API只支持基于摄像机的视图（camera-based view）模型，该模型主要是由两个参数集定义：投影属性和摄像机位置。静态视图模型的不足在于，难以进行体现视图动态变化的编程。例如，图形系统中的观察设备可能是带有头部跟踪系统的头戴式摄像机，头部跟踪（head-tracking）信息传送回图形系统，引起了视图属性的持续变化。动态视图在应用程序中的实现相当复杂，而且可能是与平台有关的。Java 3D提供了多功能的视图系统，既包含传统的视图设置，也支持动态视图。物理环境的变化对视图产生的影响可以通过分离对象自动地加以体现，而无需显式地改变基本观察结构。针对各种不同的显示要求，应用程序可以使用相同的场景图。

另一个与观察相关的主题是拾取（picking）。视图投影是把3D立体对象映射为2D图像，而拾取是从投影形成的2D图像中选择3D世界空间中的物体的过程。由此可见，拾取实际上是投影的逆过程的一部分。拾取操作利用绘制的图像，方便了用户与3D场景的交互。例如，通过拾取操作，用户可以用鼠标从屏幕图像中选取一个物体，并进行旋转或者移动。Java 3D在不同层次上为拾取提供了大量的支持。

8.2 投影

生成3D场景的2D视图是通过称为投影 (projection) 的变换过程得到的。投影主要有两种类型：平行投影 (parallel projection) 和透视投影 (perspective projection)。两种投影中，都是在虚拟世界中放置一个视平面 (view plane)，投影把虚拟世界中的点映射到该平面上。为便于实现，必须对视图添加某些约束。实际上只使用视图平面上一个有限的窗口 (通常是矩形) 来绘制图像，这一窗口称为视窗 (view plate)，这类似于普通摄像机中的一帧胶卷。只需要计算投影到此观察平面上的3D空间点，就排除了那些离观察者非常近或非常远的点。这样，3D空间中只有有限的空间区域实际参与到投影中，这一空间区域称为观察平截体 (view frustum)。

平行投影通过一组固定方向的平行线，把3D空间中的点投影到视图平面上。当投影线与视平面垂直时，平行投影称为正交 (orthographic) 投影。沿着坐标轴方向的3个正交投影，就是通常所说的前视 (front-elevation)、顶视 (top-elevation) 和侧视 (side-elevation) 投影，这三个投影经常在工程绘图中使用。平行投影视图的可视范围是平行六面体 (见图8-1)。

248

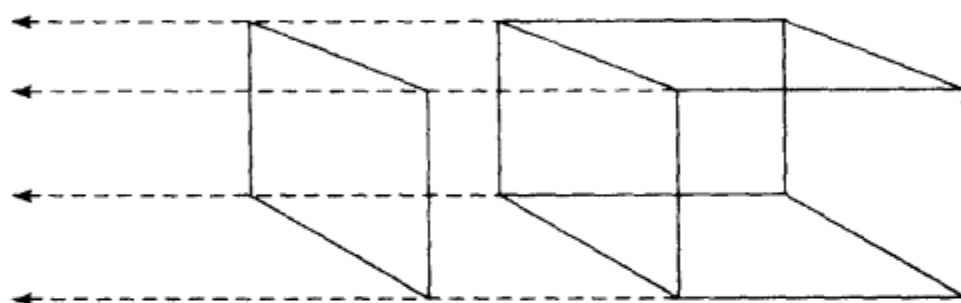


图8-1 平行投影

平行投影的计算公式相对简单。假设视平面是xy平面，投影方向是z轴正向。那么，3D空间中的点 (x, y, z) 将映射为2D视平面上的点 (x, y) ，此变换由以下矩阵方程给出：

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

事实上，为了达到确定物体遮挡部分等目的，通常需要在进行投影变换的同时保留z坐标，所以，该投影变换实质上就是一种恒等变换：

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

在透视投影中，所有的投影线汇聚于视点 (观察者或者透视投影的眼睛位置，见图8-2)。靠近观察者的物体比远离的物体看起来要大，该模型类似于人眼和普通摄像机进行的投影。

透视投影的数学公式更为复杂。假设投影平面是xy平面，眼睛在 $(0, 0, d)$ 处俯视z轴，而y轴是视图向上的方向，如图8-3所示。

249

考虑到两个相似的三角形，得到

$$\frac{y'}{d} = \frac{y}{d + (-z)}$$

或者

$$y' = \frac{y}{1 - z/d}$$

同理，在x方向上

$$x' = \frac{x}{1 - z/d}$$

所以，在3D空间中该变换不是线性的，但可用齐次坐标的线性形式来表达，参见下面的方程：

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

如果要保留z坐标，那么，变换将变成：

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

事实上，用齐次坐标和 4×4 的变换矩阵，可以用统一的方式对平行投影和透视投影进行处理。两种投影都是投影变换，可以表示为 4×4 的矩阵。显然，如果 d 趋向无穷大，那么

$$\lim_{d \rightarrow \infty} (-1/d) = 0$$

这样，上述矩阵就变为前面介绍的平行投影矩阵。平行投影可以被看做是透视投影的一个特例，其视点位于无穷远处。透视投影的视点坐标为 $(0, 0, d)$ ，或者位于齐次坐标系 $(0, 0, 1, 1/d)$ 处。当 $d \rightarrow \infty$ 时，视点接近于 $(0, 0, 1, 0)$ ，也就是齐次坐标下的无穷远点。

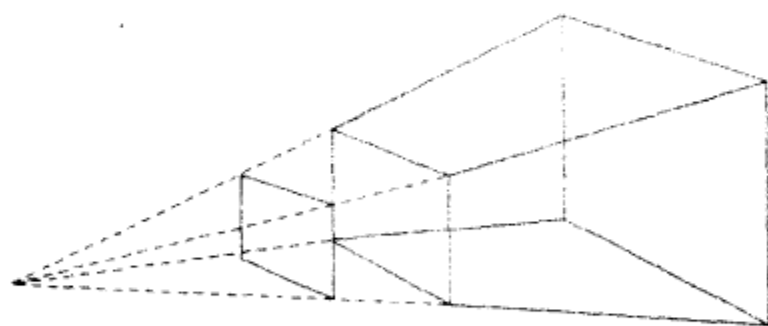


图8-2 透视投影

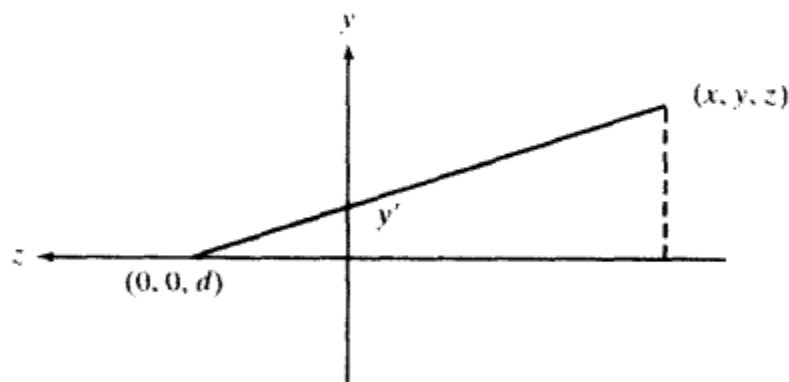


图8-3 透视投影的例子

250

8.3 视图的定义

视图的完整定义可以分为两大部分：观察体（view-volume）定义和视图定位（view positioning，其中包括位置 and 方向）。如果把视图模型看做是一台摄像机，那么观察体对应于照相机本身的一些特性（比如焦距和胶卷尺寸），而视图定位则对应于摄像机的位置和朝向。

观察体的定义通常由投影矩阵（projection matrix）来表示。视图定位由观察矩阵（viewing matrix）来表示。

下面的参数与观察体或投影矩阵的定义相关：

- 投影 (projection): 平行投影或透视投影。
- 视窗 (view plate): 绘制图像的窗口, 通常是矩形区域。在实际摄像机中, 观察平面对应于一帧胶卷。
- 视域 (field of view, fov): 平截体的左平面和右平面之间的水平角度。竖直视域和对角视域可以类似定义。
- 焦距长度 (focal length): 投影平面和视点之间的距离。
- 长宽比 (aspect ratio): 投影平面的宽度与长度之比。
- 前裁剪面 (front clip plane): 前面的或者离视点较近的平截头体的面。
- 后裁剪面 (back clip plane): 后面的或者离视点较远的平截头体的面。

上述参数并不都是相互独立的。如, 焦距长度、水平视域和投影区宽度间的关系满足以下公式:

$$\tan \frac{fov}{2} = \frac{width/2}{f}$$

拥有50mm标准镜头的35mm型照相机, 其胶卷尺寸为36mm×24mm, 观察视域为40°, 长宽比是1.5。在实际摄像机中, 胶片置于镜头之后, 所形成的投影图像是倒置的, 如图8-4所示。但是在计算机图形中, 成像胶片与图形平面恰好是对称的, 与使用倒置的图像相比, 观察平面位于眼睛前面时, 视图系统的公式推导会更为方便。

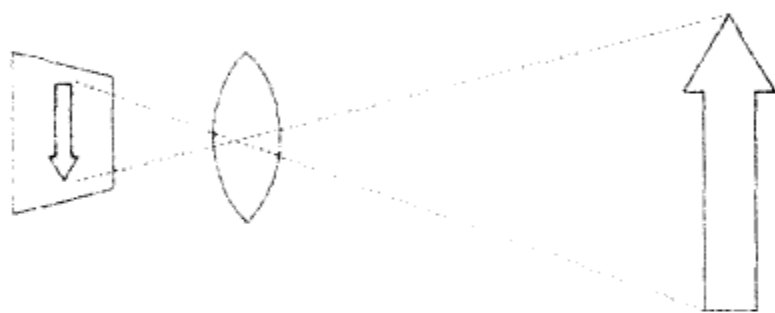


图8-4 真实的照相机投影

投影矩阵在眼睛坐标系中进行定义。在眼睛坐标系中, 眼睛位于原点, 观察方向为z轴负方向, 并且以y轴为视图的向上方向。通过定义投影变换, 实现了投影矩阵对观察体设置内容的封装, 这个投影变换把给定观察体 (平截体或立方体) 映射到标准观察体, 标准观察体通常是顶点为 $(\pm 1, \pm 1, \pm 1)$ 的立方体。例如, 对于平截头体类型的观察体而言, 其前向面顶点为 $(\pm a, \pm b, -c)$, 后向面顶点为 $(\pm a', \pm b', -d)$ 。在x和y方向上, 利用8.2节提及的参数, 投影有如下数学定义形式:

$$x' = \frac{x/a}{-z/c}$$

$$y' = \frac{y/b}{-z/c}$$

251

在z方向上, 该变换把间距 $[-d, -c]$ 映射到 $[-1, 1]$, 投影形式如下:

$$z' = \frac{1 + (1 + d/c)(z + c)/(d - c)}{-z/c}$$

因此, 该投影可以用矩阵形式表示如下:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1/a & 0 & 0 & 0 \\ 0 & 1/b & 0 & 0 \\ 0 & 0 & (1 + d/c)/(d - c) & 1 + (c + d)/(d - c) \\ 0 & 0 & -1/c & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

为在虚拟空间中完整地定义摄像机或者眼睛的定位信息 (包括位置和方向), 有必要指定摄像机的位置、摄像机所指向的方位以及摄像机所认为的“上”方向。下列参数总是与摄像机

定位或者观察矩阵 (viewing matrix) 相关联:

- 视点 (viewpoint)、视图参考点 (view-reference point, vrp, eye): 摄像机或者眼睛的位置, 摄像机所处位置的3D空间点。
- 视图中心 (view center, look): 视图平面的中心或者眼睛注视的点。
- 视图向上方向 (view up direction, up): 从观察者的视角往上看的方向。
- 视平面 (view plane): 投影图像的平面。
- 视平面法线 (view plane normal, vpn): 投影平面的法向量。

视点很容易通过一个3D点来描述。不过, 仅仅只有视点还不足以完整地定义摄像机的定位信息。显然, 在一个固定点上, 摄像机可以指向不同方向。定义3D对象在空间的方向等参数的方法有许多种, 比如说, 通常使用纵倾-横倾-横摆 (pitch-yaw-roll) 来描述飞机的定位信息。

在计算机图形学中, 定义摄像机方向的常用方法是, 指明观察参考点、观察中心和观察方向, 如图8-5所示。

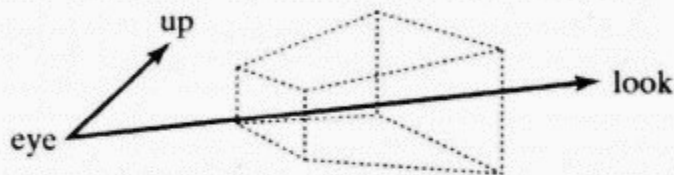


图8-5 由eye, look和up定义的观察矩阵

观察矩阵通过定义一个变换来指定摄像机的位置, 该变换用于改变摄像机的标准位置。标准摄像机位置 (观察矩阵是单位矩阵时) 是由眼睛坐标系定义的, 通常眼睛位于原点, 俯视z轴, 并且y轴指向观察者的上方。如果视点移动到 (a, b, c) 而没有改变方向, 那么观察矩阵的形式如下所示:

$$\begin{bmatrix} 1 & 0 & 0 & -a \\ 0 & 1 & 0 & -b \\ 0 & 0 & 1 & -c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

252

8.4 Java 3D的视图模型

Java 3D提供了一个功能完备的观察变换系统, 它不仅支持基于静态摄像机的传统视图功能, 同时支持基于环境变化而进行动态调整的视图功能。图8-6所示的场景图展示了视图的典型设置

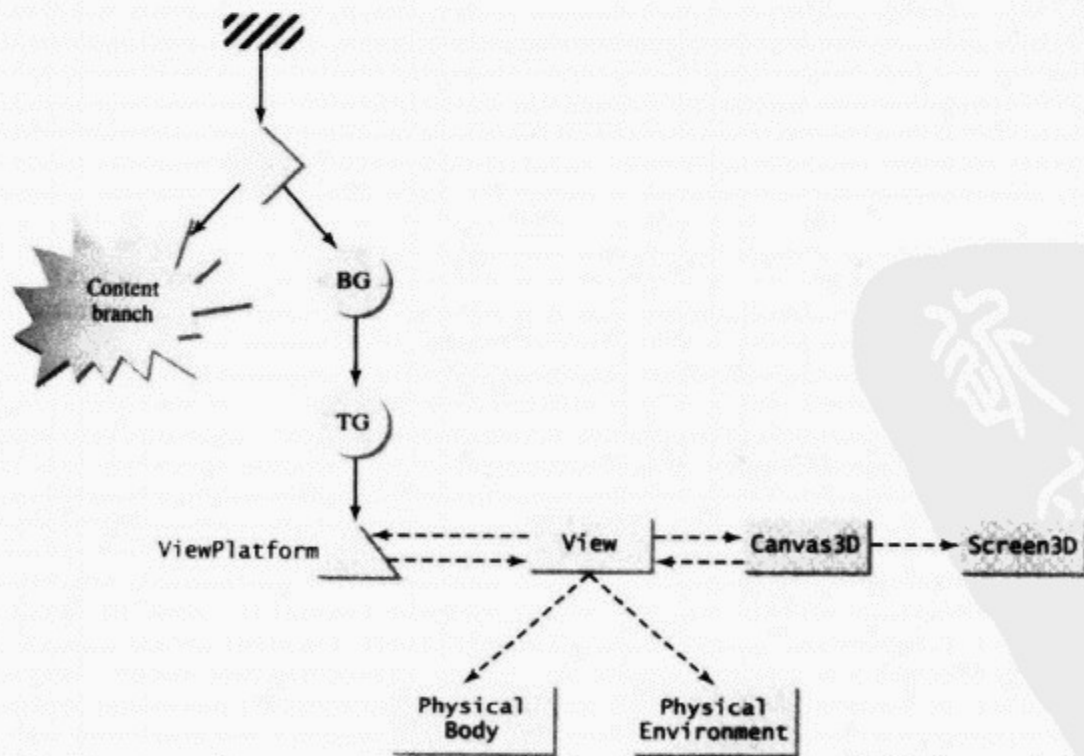


图8-6 典型的Java 3D观察系统

情形，附属在Locale对象上的，是一个定义了虚拟世界图形模型的内容分支（content branch）和一个定义了场景绘制视图的视图分支（view branch）。

Java 3D所提供的类中，直接与视图相关的类包括：ViewPlatform、View、PhysicalBody、PhysicalEnvironment、Canvas3D和Screen3D。

8.4.1 Java 3D视图配置

ViewPlatform对象是场景图的一个叶节点，它定义了虚拟世界中视图的存在。与其他场景图中的节点类似，ViewPlatform对象可以经过一系列变换节点和其他节点，最终附加到Locale对象上。连接到ViewPlatform的TransformGroup节点定义了一个视图平台变换，该变换主要确定视图的位置和方向，或者说定义了观察矩阵。观察矩阵可能受到其他因素影响，比如说，受到头部跟踪传感器输入的影响。Transform3D类中有一个方法，可帮助建立这样一种变换：

```
void lookAt(Point3D eye, Point3D look, Vector3D up) //定义视图平台变换，确定观察矩阵
```

构造这个变换的目的在于，指定ViewPlatform的位置信息，从而使眼睛位于给定点，并且注视着具有指定上方的观察平面的中心位置。这个变换的逆变换可以置于ViewPlatform上方的TransformGroup节点中，以实现ViewPlatform的指定定位信息。

253

View对象是视图系统的核心，它为具有投影类型与观察体等属性的视图定义了主要的配置内容，也就是说，定义了视图的投影矩阵。通过与PhysicalBody和PhysicalEnvironment对象的连接，它还可以针对视图参数的动态变化作出相应的调整。View类中设置观察体或投影矩阵的方法包括：

```
void setFieldofView(double fov) //设置视域
void setFrontClipDistance(double d) //设置前裁剪平面的距离
void setBackClipDistance(double d) //设置后裁剪平面的距离
void setProjectionPolicy(int projection) //设置投影类型
```

PhysicalBody和PhysicalEnvironment对象支持与动态视图系统的自动校准，比如与头盔式摄像机和头部跟踪传感器的自动校准。PhysicalBody描述了用户的身体或头部的物理特性，PhysicalEnvironment包含了关于用户的物理环境的信息，如跟踪传感器。

Canvas3D类是AWT Canvas类的子类，可以放在AWT容器中使用。Canvas3D对象表示绘制表面。下面的构造函数通常用来创建Canvas3D对象：

```
public Canvas3D(GraphicsConfiguration gc)
```

单独定义了一个Screen3D类用于描述显示设备，这个类由一个Canvas3D对象进行引用。将设备描述内容单独放置，可以避免在多个Canvas3D对象中出现同一信息的多个拷贝。Screen3D类没有公有类型的构造函数，并且其实例可以由Canvas3D类的getScreen3D方法得到。

8.4.2 兼容模式

Java 3D也提供了一个兼容模式，以支持类似于OpenGL视图规范的传统照相机模型。为了激活兼容模式（compatibility mode），可以使用下列View对象的方法：

```
public void setCompatibilityModeEnable(boolean enabled) //激活兼容模式
```

Transform3D类包含了构造观察矩阵和投影矩阵的方法，观察矩阵可以利用相同的“eye-look-up”方法来构造：

```
public void lookAt(Point3D eye, Point3D look, Vector3D up)
```

投影矩阵可以利用下面的3种不同方法之一来构造：

```
public void perspective(double fov, double aspect, double near, double far)//定义
透视投影
```

```
public void frustum(double left, double right, double bottom, double top, double
near, double far)// 定义透视投影
```

254

```
public void ortho(double left, double right, double bottom, double top, double
near, double far)// 定义平行投影
```

这些方法的坐标都是相对于眼睛位置或者vrp。perspective方法通过指定视图的水平视域、长宽比、近裁剪平面和远裁剪平面来定义一个透视投影矩阵。frustum和ortho方法都是通过指定观察体的对角坐标定义投影矩阵。(left, bottom, -near) 和 (right, top, -near) 定义了近平面的左下角和右上角，远平面的深度由-far定义。frustum方法定义透视投影，而ortho定义平行投影。

例如，下面两个调用定义了一个相同的透视投影：

```
perspective(Math.PI/2, 2, 1, 2);
frustum(-1, 1, -0.5, 0.5, 1, 2);
```

观察体的视图水平视域为 $\pi/2$ ，长宽比为2.0，近平面距离为1，远平面距离为2。近平面为 2×1 ，远平面为 4×2 。假设近平面为投影平面，则投影可以写成：

$$\begin{aligned}x' &= \frac{x}{-z} \\y' &= \frac{2y}{-z} \\z' &= \frac{3z + 4}{-z}\end{aligned}$$

于是该投影的投影矩阵为：

$$\begin{bmatrix}1 & 0 & 0 & 0 \\0 & 2 & 0 & 0 \\0 & 0 & 3 & 4 \\0 & 0 & -1 & 0\end{bmatrix}$$

利用View中的下列方法，可以为视图合理地设置投影矩阵和观察矩阵：

```
public void setVpcToEc(Transform3D viewingMatrix)//在兼容模式下设置观察矩阵
public void setLeftProjection(Transform3D projectionMatrix)//设置左眼的投影矩阵
public void setRightProjection(Transform3D projectionMatrix)//设置右眼的投影矩阵
```

在兼容模式下，几乎所有的视图参数都是在View对象中进行设置。另一方面，在标准的（非兼容）Java 3D视图模式下，最终的视图设置可能会受到诸如PhysicalBody和Physical-Environment之类的其他对象的影响。上述直接设置投影矩阵和观察矩阵的方法，在非兼容模式下并不可用。

兼容模式是一种受限的视图模式，它并不拥有所有的Java 3D视图特征，它旨在提供一个与OpenGL类型的调用相兼容的简单模式。

255

程序清单8-1以兼容模式下手工构造视图的方式，显示了一个旋转四面体，它演示了Java 3D视图所需要的最小设置内容（见图8-7）。

程序清单8-1 CompatibilityMode.java

```
1 package chapter8
2
```



```
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import chapter6.*;
10 import java.applet.*;
11 import com.sun.j3d.utils.applet.MainFrame;
12 //定义CompatibilityMode类, 继承自Applet类, 演示使用兼容模式的观察
13 public class CompatibilityMode extends Applet {
14     public static void main(String[] args) {
15         new MainFrame(new CompatibilityMode(), 640, 650); //创建主窗口并设置大小
16     }
17     //重写Applet初始化函数
18     public void init() {
19         //创建canvas
20         GraphicsConfiguration gc =
21             SimpleUniverse.getPreferredConfiguration();
22         Canvas3D cv = new Canvas3D(gc);
23         setLayout(new BorderLayout());
24         add(cv, BorderLayout.CENTER);
25         VirtualUniverse universe = new VirtualUniverse(); //创建虚拟世界对象
26         Locale locale = new Locale(universe);
27         BranchGroup bg = createView(cv); //建立场景图的观察分支
28         locale.addBranchGraph(bg); //插入到VirtualUniverse对象中
29         bg = createContent(); //建立场景图的内容分支
30         bg.compile();
31         locale.addBranchGraph(bg);
32     }
33     //生成BranchGroup的私有方法, 建立场景图中的观察分支
34     private BranchGroup createView(Canvas3D cv) {
35         BranchGroup bg = new BranchGroup();
36         ViewPlatform platform = new ViewPlatform();
37         bg.addChild(platform);
38         View view = new View();
39         view.addCanvas3D(cv);
40         view.setCompatibilityModeEnable(true); //启用兼容模式
41         view.attachViewPlatform(platform);
42         Transform3D projection = new Transform3D();
43         projection.frustum(-0.1, 0.1, -0.1, 0.1, 0.2, 10); //设置投影矩阵
44         view.setLeftProjection(projection);
45         Transform3D viewing = new Transform3D();
46         Point3d eye = new Point3d(0,0,1);
47         Point3d look = new Point3d(0,0,-1);
48         Vector3d up = new Vector3d(0,1,0);
49         viewing.lookAt(eye, look, up);
50         view.setVpcToEc(viewing); //设置观察矩阵
51         PhysicalBody body = new PhysicalBody();
52         view.setPhysicalBody(body);
53         PhysicalEnvironment env = new PhysicalEnvironment();
54         view.setPhysicalEnvironment(env);
55         return bg;
56     }
```

256

```

57 //生成BranchGroup的私有方法, 建立场景中的内容分支
58 private BranchGroup createContent() {
59     BranchGroup root = new BranchGroup();
60     TransformGroup spin = new TransformGroup();
61     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
62     root.addChild(spin);
63     //创建一个四面体
64     Appearance ap = new Appearance();
65     ap.setMaterial(new Material());
66     Shape3D shape = new Shape3D(new Tetrahedron(), ap);
67     //设置旋转变换
68     Transform3D tr = new Transform3D();
69     tr.setScale(0.25);
70     TransformGroup tg = new TransformGroup(tr);
71     spin.addChild(tg);
72     tg.addChild(shape);
73     Alpha alpha = new Alpha(-1, 4000);
74     RotationInterpolator rotator =
75     new RotationInterpolator(alpha, spin);
76     BoundingSphere bounds = new BoundingSphere();
77     rotator.setSchedulingBounds(bounds);
78     spin.addChild(rotator);
79     //设置光照和背景
80     Background background = new Background(1.0f, 1.0f, 1.0f);
81     background.setApplicationBounds(bounds);
82     root.addChild(background);
83     AmbientLight light =
84     new AmbientLight(true, new Color3f(Color.red)); // 添加环境光源
85     light.setInfluencingBounds(bounds);
86     root.addChild(light);
87     PointLight ptlight = new PointLight(new Color3f(Color.green),
88     new Point3f(3f, 3f, 3f), new Point3f(1f, 0f, 0f)); // 添加点光源
89     ptlight.setInfluencingBounds(bounds);
90     root.addChild(ptlight);
91     PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
92     new Point3f(-2f, 2f, 2f), new Point3f(1f, 0f, 0f)); // 添加另一个点光源
93     ptlight2.setInfluencingBounds(bounds);
94     root.addChild(ptlight2);
95     return root;
96 }
97 }

```

该程序直接创建场景图中的所有组件而没有使用工具类。Canvas3D、VirtualUniverse和Locale对象都在构造函数里创建。createView方法（第34行）建立场景的视图分支，而createContent建立内容分支。

方法createView构造视图分支中的对象：一个BranchGroup、一个ViewPlatform、一个View、一个PhysicalBody和一个PhysicalEnvironment对象，它激活了View对象的兼容模式（第40行）。通过调用方法frustum（第43行）将投影矩阵设置成透视投影。近裁剪平面由角点（-0.1, -0.1, -0.2）和（0.1, 0.1, -0.2）定义，远裁剪平面距离为10。观察矩阵由方法lookAt建立，眼睛位于点（0, 0, 1），视点为（0, 0, -1），以（0, 1, 0）为视图上方（第49行）。投影矩阵和观察矩阵两者都直接用View对象中的方法setLeftProjection和setVpcToEc来设定。虽然头部跟踪在兼容模式下不可用，但是仍然需要为View对象提供PhysicalBody和PhysicalEnvironment对象。

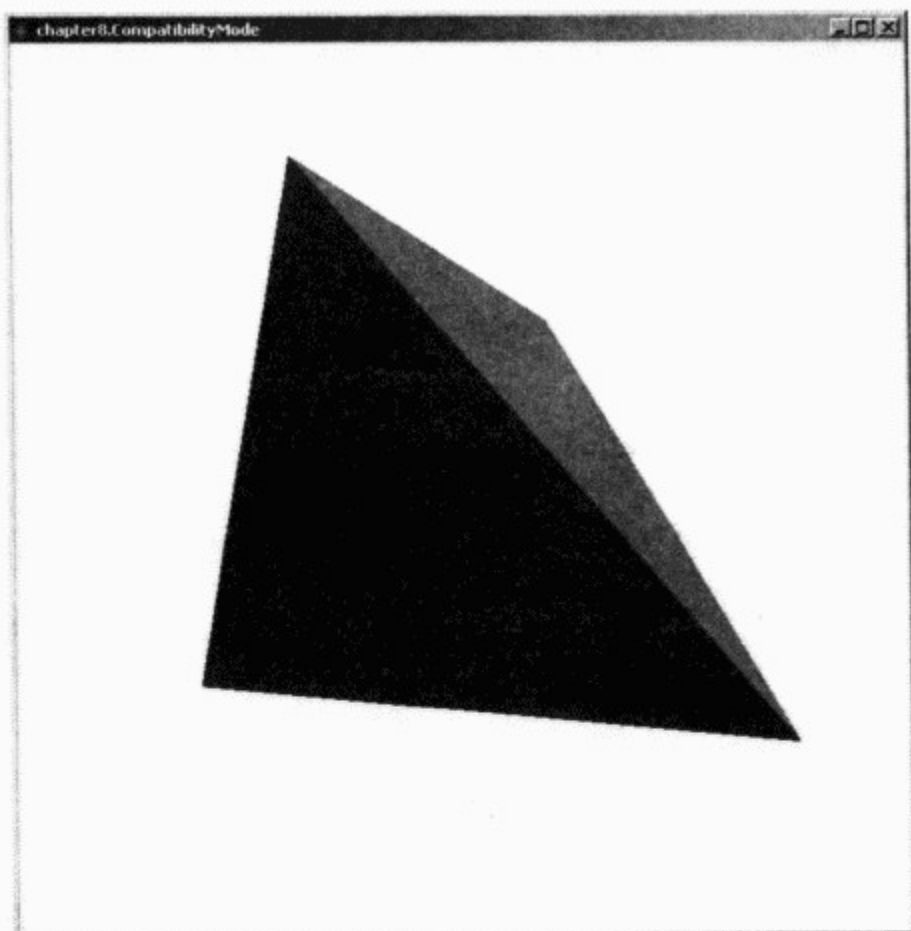


图8-7 使用兼容模式的视图

方法createContent构建场景图的内容分支，它包含了类似于程序清单6-1中所用到的旋转四面体。

257

8.4.3 SimpleUniverse中的视图设置

如图8-4所示，工具类SimpleUniverse提供了视图系统的一个默认实现。SimpleUniverse进一步把与视图相关的组件对象分成两大类：Viewer类和ViewingPlatform类。Viewer对象由一个View对象、一个ViewAvatar对象、一个PhysicalBody对象、一个PhysicalEnvironment对象和一组Canvas3D对象组成。ViewingPlatform对象包括一个ViewPlatform对象以及一个由一系列链接起来的TransformGroup节点组成的MultiTransformGroup对象。默认的ViewingPlatform在其MultiTransform Group对象中存在着一个TransformGroup对象，但是SimpleUniverse具有指定多个TransformGroup节点的构造函数：

```
SimpleUniverse(int numTrans)
SimpleUniverse(Canvas3D canvas, int numTrans)
```

为了从SimpleUniverse对象获取View，可以使用下列语句：

```
View view = su.getViewer().getView();
```

为了得到ViewPlatform之上的TransformGroup节点，可以使用下列方法调用：

```
TransformGroup tg = su.getViewingPlatform().getViewPlatformTransform();
```

258

为了在MultiTransformGrouop对象中获得一个特定的TransformGroup，可使用下列调用：

```
TransformGroup tg =
su.getViewingPlatform().getMultiTransformGroup().getTransformGroup(idx);
```

SimpleUniverse对象的默认视图设置如下：

- 兼容模式：没有激活。
- 左投影矩阵：单位矩阵。

- 右投影矩阵：单位矩阵。
- vpc-to-ec变换：单位矩阵。
- 视域： $\pi/4$ 。
- 前裁剪平面距离：0.1。
- 后裁剪平面距离：10。

下面的代码片段改变了SimpleUniverse对象中的默认设置，将ViewPlatform移动到(1, 1, 1)并注视着原点，视域设置为 0.4π ：

```
SimpleUniverse su = new SimpleUniverse(cv);
TransformGroup tg =
    su.getViewingPlatform().getMultiTransformGroup().getTransformGroup(0); // 获取TransformGroup节点
Transform3D tx = new Transform3D();
tx.lookAt(new Point3D(1, 1, 1), new Point3D(0, 0, 0), new Vector3D(0, 1, 0)); // 设置视图平台变换，移动到(1, 1, 1)，以y轴为视图上方，并注视着原点
tx.invert();
tg.setTransform(tx);
View view = su.getViewer().getView();
view.setFieldOfView(0.4*Math.PI); // 改变视域
```

程序清单8-2演示了SimpleUniverse的应用情况，以及标准Java 3D视图的控制效果。该例子显示了旋转视图下的十二面体，内容分支的可视对象即十二面体位于固定位置。不过，将视图设置为绕y轴旋转，使用户可以看到该十二面体的不同侧面（见图8-8）。

程序清单8-2 RotateView.java

```
1 package chapter8;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import chapter6.*;
10 import java.applet.*;
11 import com.sun.j3d.utils.applet.MainFrame;
12 //定义RotateView类，继承自Applet类，演示旋转视图
13 public class RotateView extends Applet {
14     public static void main(String[] args) {
15         new MainFrame(new RotateView(), 640, 480); // 创建主窗口并设定大小
16     }
17     //重写Applet初始化函数
18     public void init() {
19         //创建canvas
20         GraphicsConfiguration gc =
21             SimpleUniverse.getPreferredConfiguration();
22         Canvas3D cv = new Canvas3D(gc);
23         setLayout(new BorderLayout());
24         add(cv, BorderLayout.CENTER);
25         SimpleUniverse su = new SimpleUniverse(cv, 2); // 创建并设置SimpleUniverse
26         su.getViewingPlatform().setNominalViewingTransform();
```



```
27     BranchGroup bg = createSceneGraph(su.getViewingPlatform().
28         getMultiTransformGroup().getTransformGroup(0));
29     bg.compile();
30     su.addBranchGraph(bg); //建立场景图并插入到SimpleUniverse对象中
31 }
32 //生成BranchGroup的私有方法, 创建场景图
33 private BranchGroup createSceneGraph(TransformGroup vtg) {
34     BranchGroup root = new BranchGroup();
35     //创建一个十二面体
36     Appearance ap = new Appearance();
37     ap.setMaterial(new Material());
38     Shape3D shape = new Dodecahedron();
39     shape.setAppearance(ap);
40     Transform3D tr = new Transform3D();
41     tr.setScale(0.25);
42     TransformGroup tg = new TransformGroup(tr);
43     root.addChild(tg);
44     tg.addChild(shape);
45     //旋转
46     Alpha alpha = new Alpha(-1, 4000);
47     RotationInterpolator rotator =
48     new RotationInterpolator(alpha, vtg);
49     BoundingSphere bounds = new BoundingSphere();
50     rotator.setSchedulingBounds(bounds);
51     root.addChild(rotator);
52     //设置背景和光照
53     Background background = new Background(1.0f, 1.0f, 1.0f);
54     background.setApplicationBounds(bounds);
55     root.addChild(background);
56     AmbientLight light = new AmbientLight
57         (true, new Color3f(Color.red)); //添加环境光源
58     light.setInfluencingBounds(bounds);
59     root.addChild(light);
60     PointLight ptlight = new PointLight(new Color3f(Color.green),
61         new Point3f(3f, 3f, 3f), new Point3f(1f, 0f, 0f)); //添加绿色点光源
62     ptlight.setInfluencingBounds(bounds);
63     root.addChild(ptlight);
64     PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
65         new Point3f(-2f, 2f, 2f), new Point3f(1f, 0f, 0f)); //添加橙色点光源
66     ptlight2.setInfluencingBounds(bounds);
67     root.addChild(ptlight2);
68     return root;
69 }
70 }
```

场景图如8-9所示。该程序与程序清单6-3很相似, 但是在本例中, 旋转行为并没有应用到可视对象(十二面体)上, 而是应用到了视图上(第48行), 所以物体看上去是绕着相反方向旋转的。由于光线没有移动, 因而各表面的光照并没有随着视图旋转而改变, 我们甚至可以看到光源照不到的对象的背面。

SimpleUniverse是利用两个TransformGroup节点建立的(第25行), 其中一个节点由setNominal-ViewTransform方法使用, 以向后移动视图, 而另一个变换节点则是旋转的目标。

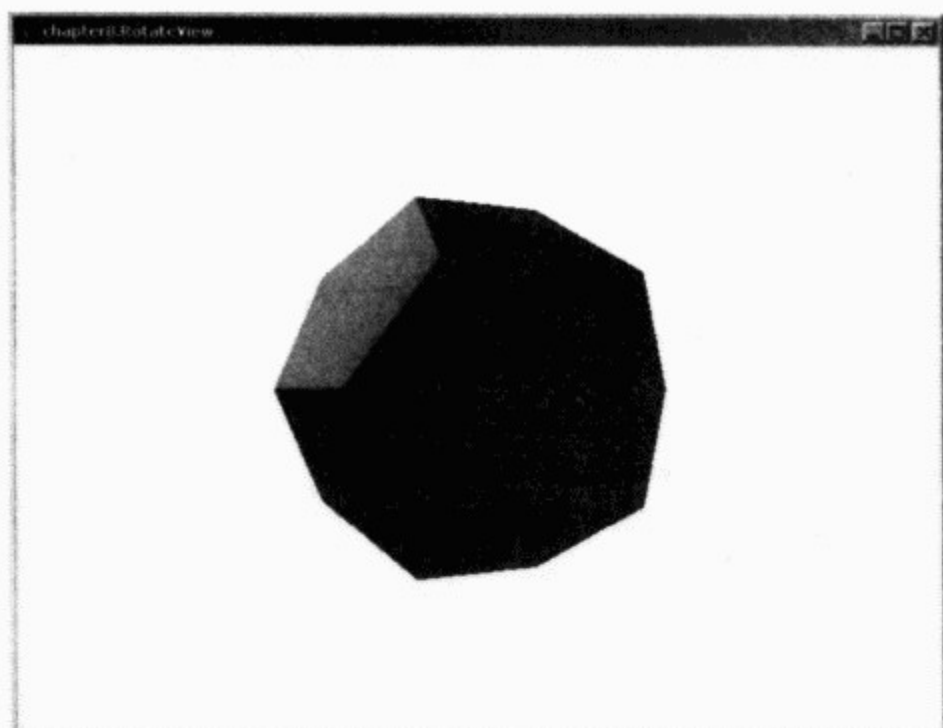


图8-8 旋转视图可以使用户看到对象的背面

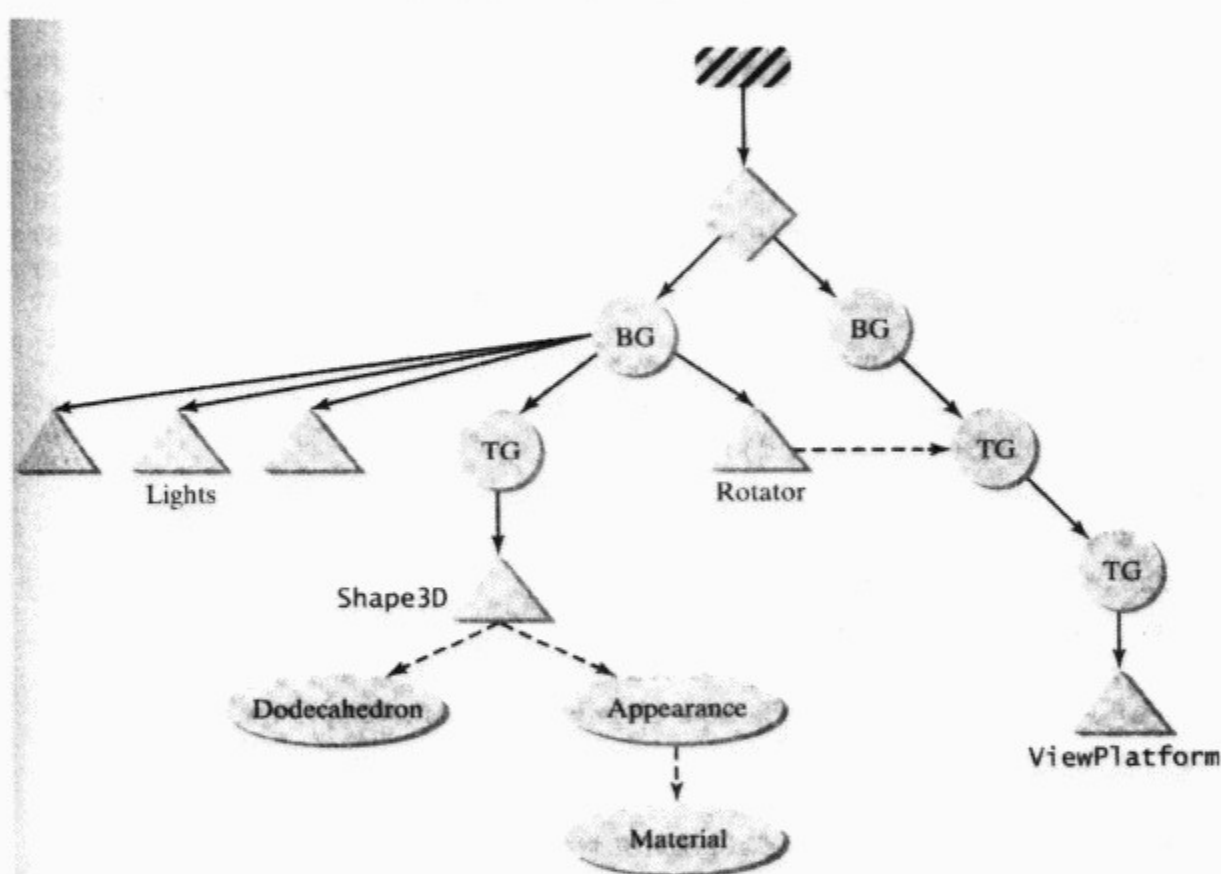


图8-9 旋转视图的场景图

8.4.4 建立自己的视图

虽然SimpleUniverse为许多Java 3D应用提供了一条构造视图分支的便捷方法，但这在某些情况下还是显得不够。我们总是可以通过设置与视图相关的合适对象来手工地创建视图分支，与视图相关的对象有View、ViewPlatform、PhysicalBody和PhysicalEnvironment等。下面的代码描述了手工创建视图分支的步骤：

```
View view = new View();
view.setProjectionPolicy(View.PARALLEL_PROJECTION); // 设置为平行投影
ViewPlatform vp = new ViewPlatform();
view.addCanvas3D(cv);
view.attachViewPlatform(vp);
```



```
view.setPhysicalBody(new PhysicalBody());
view.setPhysicalEnvironment(new PhysicalEnvironment());
Transform3D trans = new Transform3D();
trans.lookAt(eye, center, vup);//设置视图平台变换, 确定观察矩阵
trans.invert();
TransformGroup tg = new TransformGroup(trans);
tg.addChild(vp);
BranchGroup bgView = new BranchGroup();
bgView.addChild(tg);
```

一个场景图可能包含多个视图。单个场景图的多个视图可以使你从不同角度来观察同一个虚拟世界。

程序清单8-3给出了一个同时使用多个视图的应用实例, 该程序用四个不同的视图来绘制同一个对象 (一个旋转的3D文本)。其中的一个视图是SimpleUniverse提供的标准透视视图, 另外三个视图分别是x、y和z方向上的平行投影视图 (见图8-10)。

程序清单8-3 MultipleViews.java

```
1 package chapter8;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义MultipleViews类, 继承自Applet, 演示多种视图的效果
12 public class MultipleViews extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new MultipleViews(), 640, 480);//创建主窗口并设置大小
15     }
16     //重写Applet的初始化函数
17     public void init() {
18         //创建4个Canvas3D对象
19         this.setLayout(new GridLayout(2,2));
20         GraphicsConfiguration gc =
21             SimpleUniverse.getPreferredConfiguration();
22         //标准视图
23         Canvas3D cv = new Canvas3D(gc);
24         add(cv);
25         SimpleUniverse su = new SimpleUniverse(cv);
26         su.getViewingPlatform().setNominalViewingTransform();
27         // x方向的视图
28         cv = new Canvas3D(gc);
29         add(cv);
30         BranchGroup bgView = createView(cv, new Point3d(2.7,0,0),
31             new Point3d(0,0,0), new Vector3d(0,1,0));
32         su.addBranchGraph(bgView);
33         // z方向的视图
34         cv = new Canvas3D(gc);
35         add(cv);
36         bgView = createView(cv, new Point3d(0, 0, 2.7),
```

```

37     new Point3d(0,0,0), new Vector3d(0,1,0));
38     su.addBranchGraph(bgView);
39     // y方向的视图
40     cv = new Canvas3D(gc);
41     add(cv);
42     bgView = createView(cv, new Point3d(0,2.7,0),
43         new Point3d(0,0,0), new Vector3d(0,0,1));
44     su.addBranchGraph(bgView);
45     //创建视图分支,并插入到分支图中
46     BranchGroup bg = createSceneGraph();
47     bg.compile();
48     su.addBranchGraph(bg);
49 }
50 //生成BranchGroup的私有函数,建立场景图中的视图分支
51 private BranchGroup createView(Canvas3D cv, Point3d eye,
52     Point3d center, Vector3d vup) { //创建一个平行视图所必需的对象
53     View view = new View();
54     view.setProjectionPolicy(View.PARALLEL_PROJECTION);
55     ViewPlatform vp = new ViewPlatform();
56     view.addCanvas3D(cv);
57     view.attachViewPlatform(vp);
58     view.setPhysicalBody(new PhysicalBody());
59     view.setPhysicalEnvironment(new PhysicalEnvironment());
60     Transform3D trans = new Transform3D();
61     trans.lookAt(eye, center, vup); //确定投影矩阵
62     trans.invert();
63     TransformGroup tg = new TransformGroup(trans);
64     tg.addChild(vp);
65     BranchGroup bgView = new BranchGroup();
66     bgView.addChild(tg);
67     return bgView;
68 }
69 //生成BranchGroup的私有方法,创建场景图
70 private BranchGroup createSceneGraph() {
71     BranchGroup root = new BranchGroup();
72     TransformGroup spin = new TransformGroup();
73     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
74     root.addChild(spin);
75     //创建一个文本对象
76     Font3D font = new Font3D(new Font("Serif", Font.PLAIN, 1),
77         new FontExtrusion()); //定义字体
78     Text3D text = new Text3D(font, "Java");
79     Appearance ap = new Appearance();
80     ap.setMaterial(new Material());
81     Shape3D shape = new Shape3D(text, ap);
82     Transform3D tr = new Transform3D();
83     tr.setTranslation(new Vector3f(-1f, -0.25f, 0f));
84     TransformGroup tg = new TransformGroup(tr);
85     spin.addChild(tg);
86     tg.addChild(shape);
87     //设置旋转
88     Alpha alpha = new Alpha(-1, 24000);
89     RotationInterpolator rotator = new RotationInterpolator
90     (alpha, spin);

```



```
91 BoundingSphere bounds = new BoundingSphere();
92 rotator.setSchedulingBounds(bounds);
93 spin.addChild(rotator);
94 //设置背景和光照
95 Background background = new Background(1.0f, 1.0f, 1.0f);
96 background.setApplicationBounds(bounds);
97 root.addChild(background);
98 AmbientLight light = new AmbientLight
99     (true, new Color3f(Color.red)); //添加红色环境光源
100 light.setInfluencingBounds(bounds);
101 root.addChild(light);
102 PointLight ptlight = new PointLight(new Color3f(Color.green),
103     new Point3f(3f,3f,3f), new Point3f(1f,0f,0f)); //添加绿色点光源
104 ptlight.setInfluencingBounds(bounds);
105 root.addChild(ptlight);
106 PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
107     new Point3f(-2f,2f,2f), new Point3f(1f,0f,0f)); //添加橙色点光源
108 ptlight2.setInfluencingBounds(bounds);
109 root.addChild(ptlight2);
110 return root;
111 }
```

263

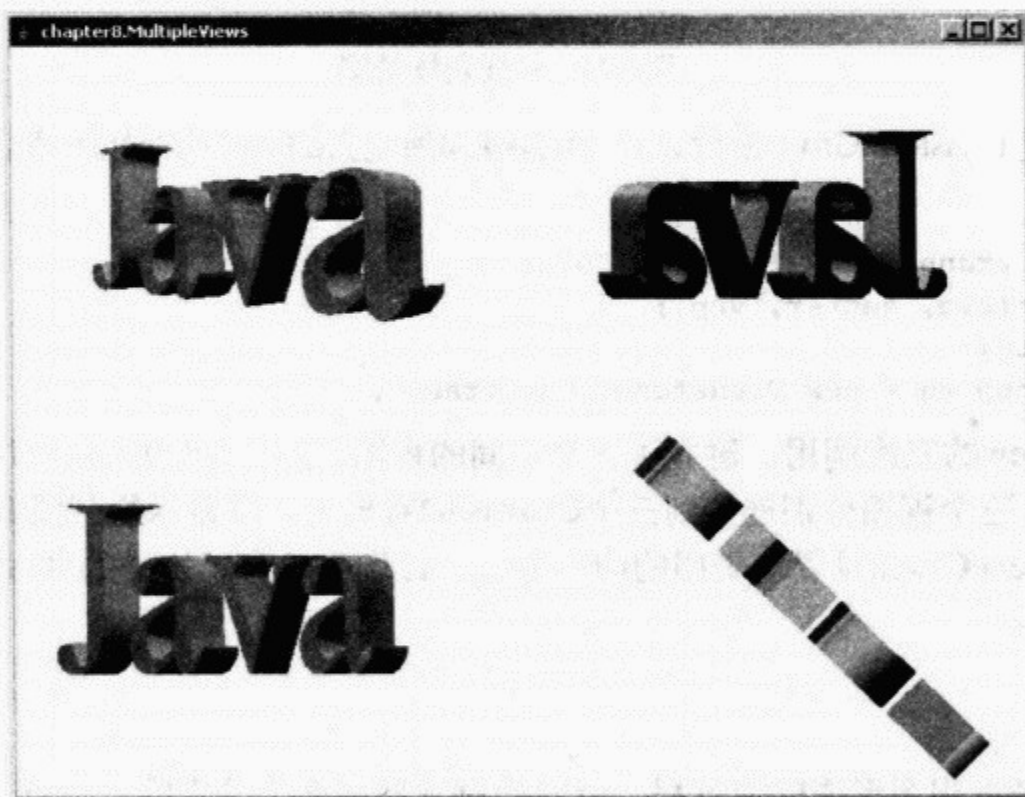


图8-10 一个场景的四个视图

此程序在同一个场景图中建立四个不同的视图，其中一个视图是在SimpleUniverse对象中通过透视投影得到的标准视图，其他三个视图分别是沿着x、y和z轴的平行投影视图。于是，它们分别对应于工程制图里的正视图、俯视图和侧视图。

图8-11显示了该例子的场景图。在网格中放置四个Canvas3D对象，方法createView（第51行）用于建立平行视图所必需的对象。每一个视图都拥有单独的BranchGroup、TransformGroup、ViewPlatform、View、PhysicalBody和PhysicalEnvironment对象实例。投影的类型用View的下列方法进行设置：

```
view.setProjectionPolicy(View.PARALLEL_PROJECTION);
```

264

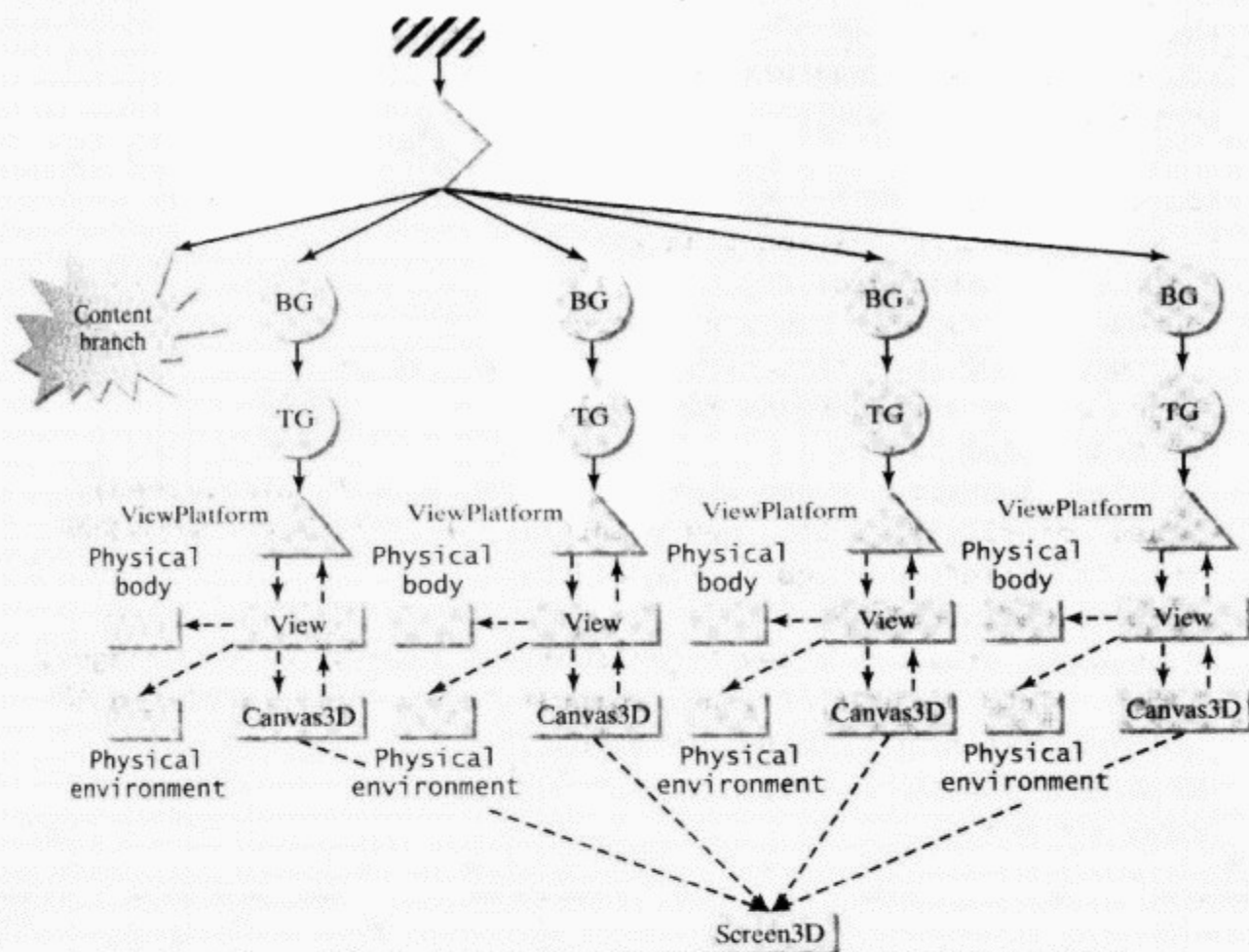


图8-11 多视角场景图

观察矩阵通过TransformGroup来设置。方法lookAt所定义的矩阵的逆矩阵适合于TransformGroup使用：

```
Transform3D trans = new Transform3D();
trans.lookAt(eye, center, vup);
trans.invert();
TransformGroup tg = new TransformGroup(trans);
```

方法createView的三次调用，建立了三个不同的视图，每一个以BranchGroup为根的视图都附加到Locale上。三个视图分别连接到三个Canvas3D对象上，以显示绘制图像。

方法createSceneGraph设置场景图的内容分支，该场景包含了旋转的3D文本“Java”、背景对象和光源对象。

8.5 拾取

有时也需要考虑视图形成的逆过程。针对绘制观察平面上给定的一个点，要确定投影到该点的可视对象，这就是所谓的拾取（picking）过程。拾取的一个典型应用，是让用户使用鼠标选择绘制图像中的物体。

显然，在投影变换下，2D观察平面上的一个点并不与3D虚拟世界中的单个点相对应，一条射线上的点都会投影到该点。如果比较现实地把点看做是一个半径取值为正的圆盘，让它有一定的余量而不是一个理想的点，那么该点在投影前的图像集合对应一个圆锥体（在透视投影下）或者是圆柱体（在平行投影下）。

265

考虑图8-3的简单投影。如果点 (x', y') 在观察平面上，拾取射线上的点 (x, y, z) 满足方程

$$\begin{aligned} x &= x'(1 - z/d) \\ y &= y'(1 - z/d) \end{aligned}$$

如果选择了参数 $t = 1 - z/d$ ，那么拾取射线的方程为

$$\begin{aligned}x &= x' t \\y &= y' t \\z &= d(1 - t)\end{aligned}$$

如果把点看做是半径为 r 的圆，那么拾取圆锥体有如下方程

$$\left(\frac{x}{1 - z/d} - x'\right)^2 + \left(\frac{y}{1 - z/d} - y'\right)^2 = r^2$$

Java 3D为拾取操作提供了多层次的支持功能，核心的拾取功能由 PickShape类、SceneGraphPath类、BranchGroup类以及Locale类的方法来提供支持。工具包com.sun.j3d.utils.picking和com.sun.j3d.utils.picking.behavior中的类提供了拾取的高层支持。

基本的拾取操作，通过创建PickShape对象并调用类BranchGroup或Locale中的拾取方法来执行，拾取的结果作为Scene GraphPath对象返回。

PickShape类族定义了多个拾取形状，图8-12给出了拾取形状的层次结构。

类BranchGroup和Locale包含了下述方法，可以执行拾取操作：

```
SceneGraphPath[] pickAll(PickShape pickShape)
SceneGraphPath[] pickAllSorted(PickShape pickShape)
SceneGraphPath pickAny(PickShape pickShape)
SceneGraphPath pickClosest(PickShape pickShape)
```

SceneGraphPath对象代表了场景图中从Locale对象到终端节点的一条路径，SceneGraphPath可用于拾取，因为在拾取操作的结果中，通常需要知道整个路径而不仅仅是要拾取的节点。

下面的代码片段大致描述了对一个BranchGroup进行拾取操作的过程：

```
Point3d origin = new Point3d(0, 0, 0); // 视点
Vector3d direction = new Vector3d(0, 1, -1); // 视图方向
PickShape pickShape = new PickRay(origin, direction);
SceneGraphPath[] paths = branchGroup.pickAll(pickShape); // 拾取所有与拾取形体相交的孩子对象
for (int i = 0; i < paths.length; i++) {
    <operations on paths[i]...> // 对每一个对象分别进行处理
}
```

拾取操作最普遍的应用是让用户在一个绘制的图像上选择对象。为了简化这一过程，工具包com.sun.j3d.utils.picking包含了四个类，可以在场景图的分支上进行一般的拾取操作（见图8-13）。

PickTool类提供了一种方便的方法，可以设置拾取形状，并对场景图的分支进行拾取操作，它有如下的拾取方法：

```
PickResult[] pickAll()
PickResult[] pickAllSorted()
PickResult pickAny();
PickResult pickClosest()
```

子类PickCanvas通过与画布的关联，简化了拾取形状的设置，可以使用鼠标事件自动生成拾取形状。下面的PickCanvas的构造函数，建立了与一个Canvas3D对象和一个BranchGroup对象的连接。拾取操作将作用于此场景图分支，拾取形状可以基于画布上的鼠标事件形成：

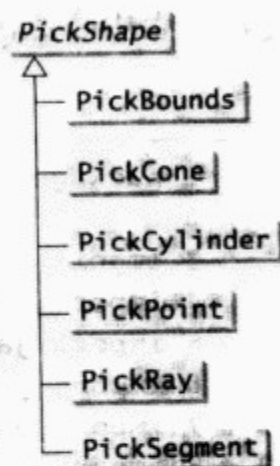


图8-12 PickShape的类层次结构

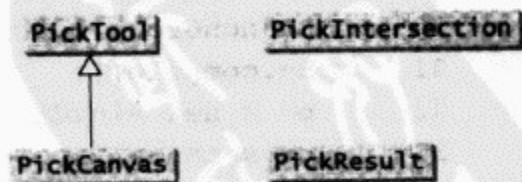


图8-13 Picking类


```
public pickCanvas(Canvas3D cv, BranchGroup bg)
```

对象Locale同样有类似的构造函数:

```
public pickCanvas(Canvas3D cv, Locale lc)
```

拾取操作的结果在对象PickResult中返回, PickResult包含了诸如被拾取的节点对象、SceneGraphPath和相交等信息。类PickIntersection包含了更多关于拾取形状和被拾取节点的相交情况的细节信息。

程序清单8-4演示了拾取功能类的一个简单应用。本例的场景由六个几何基元组成:一个球、一个立方体、一个圆柱体、一个圆锥体、一个四面体和一个十二面体,它们以线框形式绕y轴旋转。当用户点击其中的一个几何基元时,物体将会呈现受到光照的彩色外观(见图8-14)。

程序清单8-4 Picking.java

```
1 package chapter8;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import com.sun.j3d.utils.picking.*;
10 import chapter6.*;
11 import java.applet.*;
12 import com.sun.j3d.utils.applet.MainFrame;
13 //定义Picking类,继承自Applet类,实现MouseListener接口,演示拾取操作
14 public class Picking extends Applet implements MouseListener{
15     public static void main(String[] args) {
16         new MainFrame(new Picking(), 640, 480); //创建主窗口并设定窗口大小
17     }
18
19     PickCanvas pc; //声明PickCanvas变量
20     Appearance lit = new Appearance(); //声明并初始化外观对象
21     //重写Applet的初始化函数
22     public void init() {
23         //创建canvas
24         GraphicsConfiguration gc =
25             SimpleUniverse.getPreferredConfiguration();
26         Canvas3D cv = new Canvas3D(gc);
27         setLayout(new BorderLayout());
28         add(cv, BorderLayout.CENTER);
29         cv.addMouseListener(this);
30         BranchGroup bg = createSceneGraph(); //调用createSceneGraph建立场景图
31         bg.compile();
32         pc = new PickCanvas(cv, bg); //初始化PickCanvas对象
33         pc.setMode(PickTool.GEOMETRY); //设定模式
34         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
35         su.getViewingPlatform().setNominalViewingTransform();
36         su.addBranchGraph(bg);
37     }
38     //生成BranchGroup的私有方法,用于创建场景图
39     private BranchGroup createSceneGraph() {
40         BranchGroup root = new BranchGroup();
41         TransformGroup spin = new TransformGroup();
```



```
42 spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
43 root.addChild(spin);
44 //设置外观
45 Appearance wireframe = new Appearance();
46 wireframe.setPolygonAttributes(new PolygonAttributes(
47     PolygonAttributes.POLYGON_LINE,
48     PolygonAttributes.CULL_BACK, 0f));
49 wireframe.setColoringAttributes(new ColoringAttributes(
50     0f, 0f, 0f, ColoringAttributes.SHADE_FLAT));
51 lit.setMaterial(new Material());
52 //创建几何基元: 长方体、球、圆柱体、圆锥体
53 Box box =
54 new Box(1.2f, 0.3f, 0.8f, Primitive.ENABLE_GEOMETRY_PICKING |
55     Primitive.ENABLE_APPEARANCE_MODIFY |
56     Primitive.GENERATE_NORMALS, wireframe); //长方体
57 Sphere sphere = new Sphere
58     (1f, Primitive.ENABLE_GEOMETRY_PICKING |
59     Primitive.ENABLE_APPEARANCE_MODIFY |
60     Primitive.GENERATE_NORMALS, wireframe); //球体
61 Cylinder cylinder = new Cylinder(1.0f, 2.0f,
62     Primitive.ENABLE_GEOMETRY_PICKING |
63     Primitive.ENABLE_APPEARANCE_MODIFY |
64     Primitive.GENERATE_NORMALS, wireframe); //圆柱体
65 Cone cone = new Cone
66     (1.0f, 2.0f, Primitive.ENABLE_GEOMETRY_PICKING |
67     Primitive.ENABLE_APPEARANCE_MODIFY |
68     Primitive.GENERATE_NORMALS, wireframe); //圆锥体
69 Transform3D tr = new Transform3D();
70 tr.setScale(0.2); //按比例缩小
71 TransformGroup tg = new TransformGroup(tr);
72 spin.addChild(tg);
73 tg.addChild(box);
74 tr.setIdentity();
75 tr.setTranslation(new Vector3f(0f, 1.5f, 0f));
76 TransformGroup tgSphere = new TransformGroup(tr); //设置球体的平移变换
77 tg.addChild(tgSphere);
78 tgSphere.addChild(sphere);
79 tr.setTranslation(new Vector3f(-1f, -1.5f, 0f));
80 TransformGroup tgCylinder = new TransformGroup(tr); //设置圆柱体的平移变换
81 tg.addChild(tgCylinder);
82 tgCylinder.addChild(cylinder);
83 tr.setTranslation(new Vector3f(1f, -1.5f, 0f));
84 TransformGroup tgCone = new TransformGroup(tr); //设置圆锥体的平移变换
85 tg.addChild(tgCone);
86 tgCone.addChild(cone);
87 Shape3D tetra = new Shape3D(new Tetrahedron(), wireframe); //创建正四面体
88 PickTool.setCapabilities(tetra, PickTool.INTERSECT_TEST);
89 tetra.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
90 tr = new Transform3D();
91 tr.setScale(0.12);
92 tr.setTranslation(new Vector3f(0f, 0f, -0.4f)); //设置四面体的平移
93 tg = new TransformGroup(tr);
94 spin.addChild(tg);
95 tg.addChild(tetra);
```

```

96 Shape3D shape = new Dodecahedron();//创建正十二面体
97 shape.setAppearance(wireframe);//设置外观属性
98 PickTool.setCapabilities(shape, PickTool.INTERSECT_TEST);
99 shape.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
100 tr = new Transform3D();
101 tr.setScale(0.12);
102 tr.setTranslation(new Vector3f(0f, 0f, 0.4f)); //设置十二面体的平移
103 tg = new TransformGroup(tr);
104 spin.addChild(tg);
105 tg.addChild(shape);
106 //设置旋转变换
107 Alpha alpha = new Alpha(-1, 4000);
108 RotationInterpolator rotator =
109     new RotationInterpolator(alpha, spin);
110 BoundingSphere bounds = new BoundingSphere();
111 rotator.setSchedulingBounds(bounds);
112 spin.addChild(rotator);
113 //设置背景和光照
114 Background background = new Background(1.0f, 1.0f, 1.0f);
115 background.setApplicationBounds(bounds);
116 root.addChild(background);
117 AmbientLight light =
118     new AmbientLight(true, new Color3f(Color.red)); //添加红色环境光源
119 light.setInfluencingBounds(bounds);
120 root.addChild(light);
121 PointLight ptlight = new PointLight(new Color3f(Color.green),
122     new Point3f(3f, 3f, 3f), new Point3f(1f, 0f, 0f)); //添加绿色点光源
123 ptlight.setInfluencingBounds(bounds);
124 root.addChild(ptlight);
125 PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
126     new Point3f(-2f, 2f, 2f), new Point3f(1f, 0f, 0f)); //添加橙色点光源
127 ptlight2.setInfluencingBounds(bounds);
128 root.addChild(ptlight2);
129 return root;
130 }
131 //鼠标事件响应处理函数,对鼠标选中的物体设置外观属性(着色)
132 public void mouseClicked(java.awt.event.MouseEvent mouseEvent) {
133     pc.setShapeLocation(mouseEvent);
134     PickResult[] results = pc.pickAll();
135     for (int i = 0; (results != null) &&
136         (i < results.length); i++) {
137         Node node = results[i].getObject();
138         if (node instanceof Shape3D) {
139             ((Shape3D)node).setAppearance(lit);
140             System.out.println(node.toString());
141         }
142     }
143 }
144 //响应鼠标进入事件
145 public void mouseEntered(java.awt.event.MouseEvent mouseEvent) {
146 }
147 //响应鼠标离开事件
148 public void mouseExited(java.awt.event.MouseEvent mouseEvent) {
149 }

```

269


```
150 //响应鼠标按下事件
151 public void mousePressed(java.awt.event.MouseEvent mouseEvent) {
152 }
153 //响应鼠标释放事件
154 public void mouseReleased(java.awt.event.MouseEvent mouseEvent) {
155 }
156 }
```

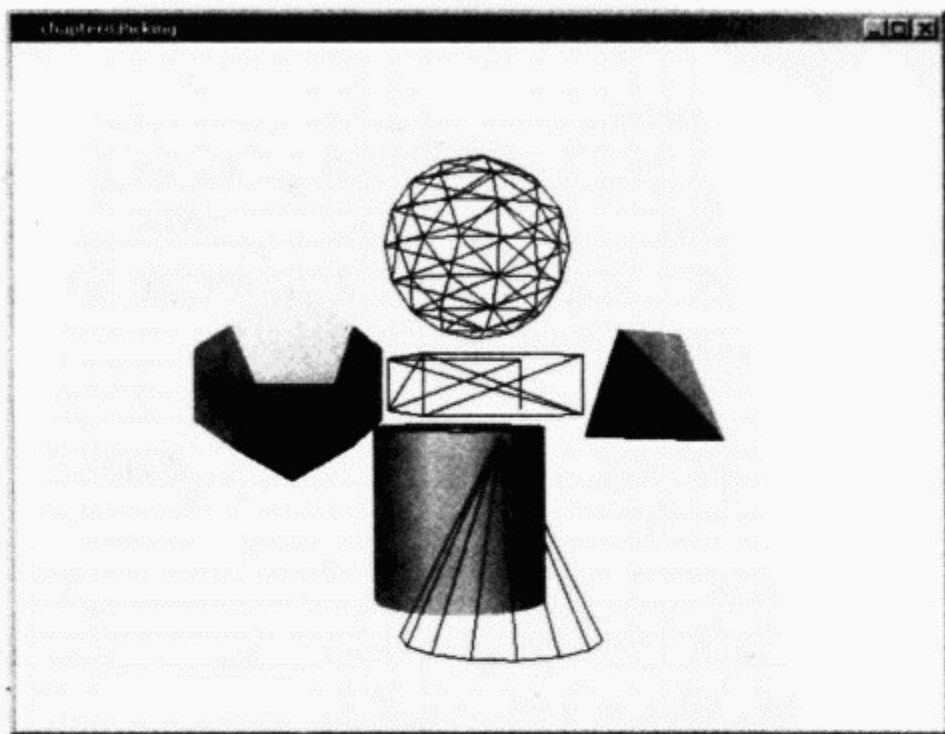


图8-14 拾取操作的演示

270

六个可视对象——一个球、一个立方体、一个圆锥体、一个圆柱体、一个四面体和一个十二面体——初始以线框形式显示。当某一个物体被鼠标点击时，其外观将会改变，变成一个接受光照的填充对象。

可视对象是以线框外观形式建立的，每一个对象都被赋予修改权限，以便可以进行拾取及改变外观。

当在对象Canvas3D上接收到鼠标事件时，创建一个PickCanvas对象（第32行）来进行拾取操作。被拾取的Shape3D对象会改变其外观，以激活光照效果。

在事件处理器mouseClicked中，根据鼠标位置设置拾取形状：

```
pc.setShapeLocation(mouseEvent);
```

调用方法pickAll()来得到所有的拾取物体（第134行），每一个被拾取的Shape3D节点的外观都会改变。

要注意的是，一个几何基元对象可以包含多个Shape3D节点，所以一个拾取操作不一定拾取所有与几何基元相关联的节点。

8.6 头部跟踪

视图的设置依赖于眼睛的位置和方向。如果观察者的头部位置改变，那么视图需要进行相应的调整。对应用程序员而言，动态地重新计算投影矩阵和观察矩阵是一件枯燥乏味的事，所以需要有一个API提供头部跟踪的功能。

Java 3D视图模型提供了一系列类，用于支持头部自动跟踪功能，这些类包括View、Sensor、PhysicalBody和PhysicalEnvironment。

头部跟踪功能的实现基础是，由某个6自由度（Six-deghee-of-freedom, 6DOF）跟踪设备提供输入信息。Java 3D提供了一个InputDevice接口，用于定义输入设备。针对特定设备实现的InputDevice接口，提供了一个Sensor数组，Sensor对象提供来自设备的输入数据。InputDevice的下列方法，用来执行设备初始化操作并获取一个Sensor对象：

```
void initialize()
Sensor getSensor(int sensorIndex)
```

PhysicalEnvironment对象拥有所有注册过的输入设备。要注册一个设备，可使用PhysicalEnvironment对象中的方法：

```
void addInputDevice(InputDevice device)
```

PhysicalEnvironment对象还维护着一个关系列表，体现6自由度实体与传感器之间的联系。如，在一个默认的PhysicalEnvironment对象中，三个预先定义的6自由度实体——UserHand、DominantHand和NondominantHand——在关系列表中的索引分别为0、1和2。为把一个传感器指派给UserHead，可以调用PhysicalEnvironment中的下述方法：

```
setSensor(0, sensor);
```

在默认情况下，视图的头部跟踪选项没有被激活。为了激活头部跟踪，要调用View中的下述方法：

```
view.setTrackingEnable(true);
```

头部跟踪视图的显示有两种不同的安装选项：空间安装（room mounted）和头部安装（head mounted）。要设置该选项，可调用View的下述方法：

```
void setViewPolicy(int viewPolicy)
```

下面的标志可以用来设置空间安装或者头部安装策略：

```
SCREEN_VIEW
HMD_VIEW
```

在头部安装模式下，有必要修改Canvas3D对象的单目视图策略，将View.CYCLOPEAN_EYE_VIEW改为View.LEFT_EYE_VIEW：

```
canvas.setMonoscopicViewPolicy(View.LEFT_EYE_VIEW)
```

观察者自身在场景中的可视形式称为替身（avatar）。SimpleUniverse对象中的Viewer对象提供了一个选项，用于包括观察者替身：

```
ViewerAvatar avatar = new ViewerAvatar();
su.getViewer().setAvatar(avatar);
```

ViewerAvatar是BranchGroup的一个子类，所以替身的形状可作为子节点加入进来。这里定义的替身位置与视图平台相关联，不一定是实际的头部位置。程序清单8-5演示了使用“虚拟输入设备”的Java 3D模型的头部跟踪功能，其中的类使用了程序清单8-6中的LineAxes类。该场景中包含了一个在背景图像之上做旋转运动的简单彩色立方体。位于一个单独窗口里的虚拟输入设备，提供了可用于控制头部位置的6自由度输入数据。由三条轴线组成的一个替身也插入到场景中，头部移动到替身后面时，就可以看到替身了。

程序清单8-5 HeadTracking.java

```
1 package chapter8;
2
3 import java.awt.*;
```



```
4 import java.awt.event.*;
5 import com.sun.j3d.utils.geometry.ColorCube;
6 import com.sun.j3d.utils.universe.*;
7 import javax.media.j3d.*;
8 import javax.vecmath.*;
9 import java.net.URL;
10 import java.awt.image.*;
11 import javax.imageio.*;
12 import java.applet.*;
13 import com.sun.j3d.utils.applet.MainFrame;
14 //定义HeadTracking类, 继承自Applet类, 用于演示头部跟踪
15 public class HeadTracking extends Applet {
16     public static void main(String[] args) {
17         new MainFrame(new HeadTracking(), 500, 500); //创建主窗口并设定大小
18     }
19     //重写Applet初始化函数
20     public void init() {
21         //创建canvas
22         GraphicsConfiguration gc =
23             SimpleUniverse.getPreferredConfiguration();
24         Canvas3D cv = new Canvas3D(gc);
25         setLayout(new BorderLayout());
26         add(cv, BorderLayout.CENTER);
27         BranchGroup scene = createSceneGraph(); //创建场景图
28         SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse对象
29         //加入替身
30         LineAxes axes = new LineAxes(0.2f);
31         Shape3D shape = new Shape3D(axes);
32         ViewerAvatar va = new ViewerAvatar();
33         va.addChild(shape);
34         su.getViewer().setAvatar(va);
35         //安装虚拟输入设备
36         String[] args = new String[0];
37         InputDevice device = new VirtualInputDevice( args );
38         device.initialize();
39         PhysicalEnvironment pe = su.getViewer().getPhysicalEnvironment();
40         pe.addInputDevice(device);
41         pe.setSensor(0, device.getSensor(0));
42         //设置头部跟踪
43         su.getViewingPlatform().setNominalViewingTransform();
44         pe.setCoexistenceCenterInPworldPolicy(View.NOMINAL_HEAD);
45         View view = su.getViewer().getView();
46         view.setUserHeadToWorldEnable(true);
47         view.setCoexistenceCenteringEnable(false);
48         Screen3D screen = cv.getScreen3D();
49         Transform3D tr = new Transform3D();
50         tr.setTranslation(new Vector3d(0.1, 0.1, 0.0)); //设定平移变换
51         screen.setTrackerBaseToImagePlate(tr);
52         view.setTrackingEnable(true); //启用跟踪
53         su.addBranchGraph(scene);
54     }
55     //生成BranchGroup的私有方法, 创建场景图
56     public BranchGroup createSceneGraph() {
```

```

57 BranchGroup objRoot = new BranchGroup();
58 TransformGroup objTrans = new TransformGroup();
59 objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
60 objRoot.addChild(objTrans);
61 objTrans.addChild(new ColorCube(0.2)); //创建一个立方体模拟头部跟踪
62 Transform3D yAxis = new Transform3D();
63 Alpha rotationAlpha = new Alpha(-1, Alpha.INCREASING_ENABLE,
64 0, 0,
65 4000, 0, 0,
66 0, 0, 0);
67 RotationInterpolator rotator = //设置旋转
68 new RotationInterpolator(rotationAlpha, objTrans, yAxis,
69 0.0f, (float) Math.PI*2.0f);
70 BoundingSphere bounds =
71 new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
72 rotator.setSchedulingBounds(bounds);
73 objTrans.addChild(rotator);
74 //装载图像作为场景图的背景
75 URL url = getClass().getClassLoader().getResource
76 ("images/bg.jpg"); //图片URL
77 BufferedImage bi = null;
78 try {
79 bi = ImageIO.read(url); //读取图片
80 } catch (Exception ex) {
81 ex.printStackTrace();
82 }
83 ImageComponent2D image =
84 new ImageComponent2D(ImageComponent2D.FORMAT_RGB, bi);
85 Background background = new Background(image);
86 background.setApplicationBounds(bounds);
87 objRoot.addChild(background);
88 return objRoot;
89 }
90 }

```

程序清单8-6 LineAxes.java

```

1 package chapter8;
2
3 import javax.media.j3d.*;
4 import javax.vecmath.*;
5 //定义LineAxes类, 继承自LineArray类, 用于定义坐标轴
6 public class LineAxes extends LineArray{
7     public LineAxes(float len) {
8         super(6, LineArray.COORDINATES | LineArray.COLOR_3);
9         setCoordinate(0, new Point3f(-len,0f,0f)); //定义点坐标值
10        setCoordinate(1, new Point3f(len,0,0f));
11        setCoordinate(2, new Point3f(0f,-len,0f));
12        setCoordinate(3, new Point3f(0f,len,0f));
13        setCoordinate(4, new Point3f(0f,0f,-len));
14        setCoordinate(5, new Point3f(0f,0f,len));
15        Color3f c0 = new Color3f(0f, 0f, 0f); //定义颜色值
16        Color3f c1 = new Color3f(1f, 0f, 0f);

```



```

17     Color3f c2 = new Color3f(0f, 1f, 0f);
18     Color3f c3 = new Color3f(0f, 0f, 1f);
19     setColor(0, c0); //设定点的颜色
20     setColor(1, c1);
21     setColor(2, c0);
22     setColor(3, c2);
23     setColor(4, c0);
24     setColor(5, c3);
25 }
26 }

```

为了演示带有头部跟踪功能的视图，需要使用一个头部跟踪设备。Java 3D包的演示程序包括一个VirtualInputDevice类（依赖于其他三个类：PositionControls、RotationControls和WheelControls），它通过一个单独的控制窗口来提供6自由度输入数据，实现了InputDevice接口。如图8-15所示的控制窗口，允许在三个方向上移动以及绕三根轴线旋转。本例中使用VirtualInputDevice来避免对特殊硬件设备的依赖（第37行）。

274

图8-16显示了该场景图。图形内容分支包含了一个在背景图像之上做旋转运动的立方体。视图分支由SimpleUniverse构建。插入一个ViewerAvatar对象，以帮助显示头部跟踪的可视化效果（第29~34行）。替身是一组由几何类LineAxes定义的坐标轴，LineAxes是LineArray的子类，它定义了三条着色的线段。

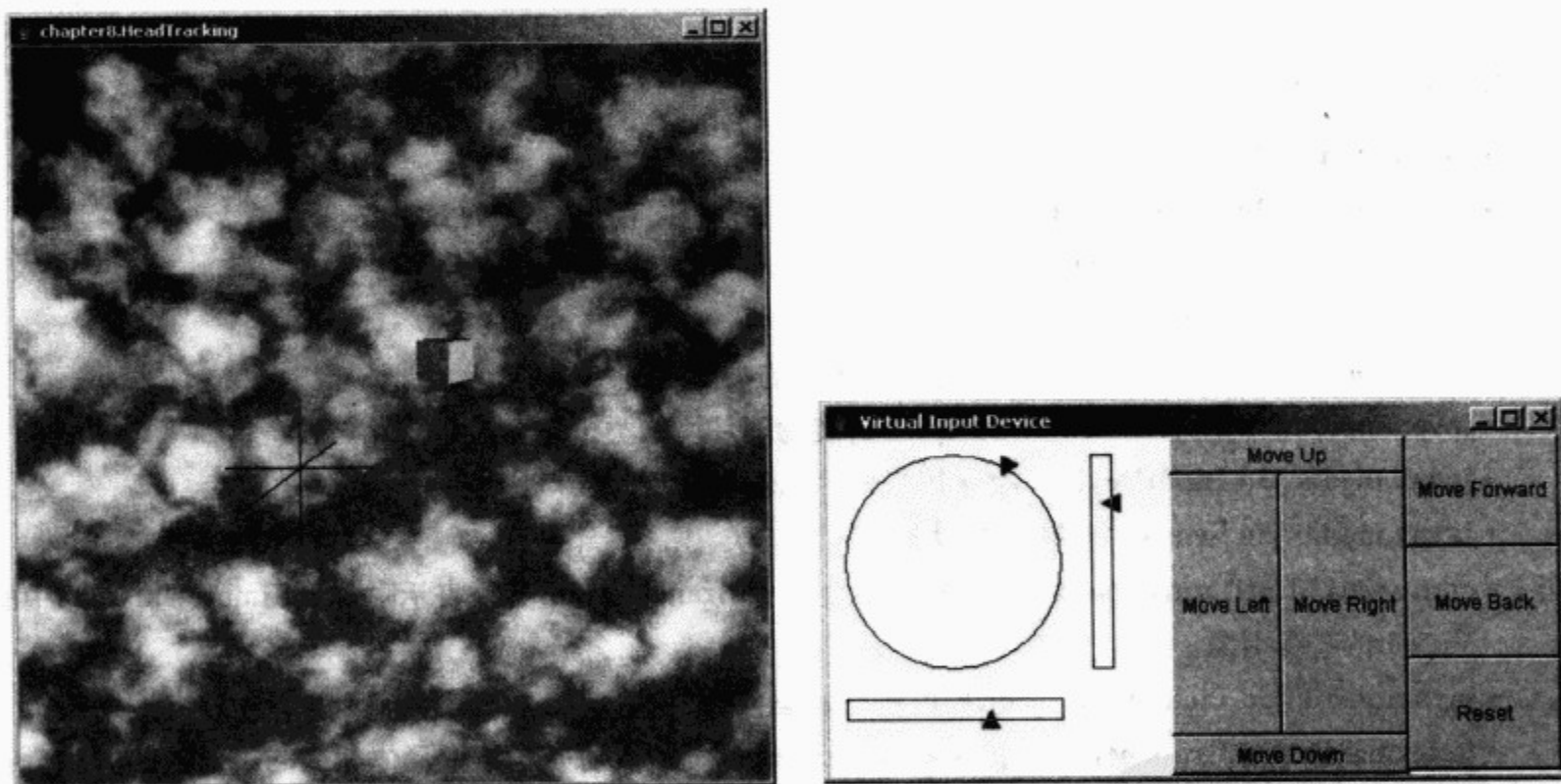


图8-15 头部跟踪的例子，虚拟输入设备为头部跟踪视图提供了模拟的跟踪输入

程序将一个VirtualInputDevice实例添加到PhysicalEnvironment之中（第40行），从VirtualInputDevice得到第一个Sensor对象，并且在PhysicalEnvironment中将它指定为UserHead实体的传感器。通过调用View中的setTrackingEnable方法，激活头部跟踪功能（第52行）。

当激活头部跟踪功能之后，默认屏幕坐标系的原点位于物理屏幕的左下角，这通常是不方便的。为了改变原点的位置，可调用Screen3D的setTrackerBaseToImagePlate (Transform3D)方法（第51行）执行平移变换。

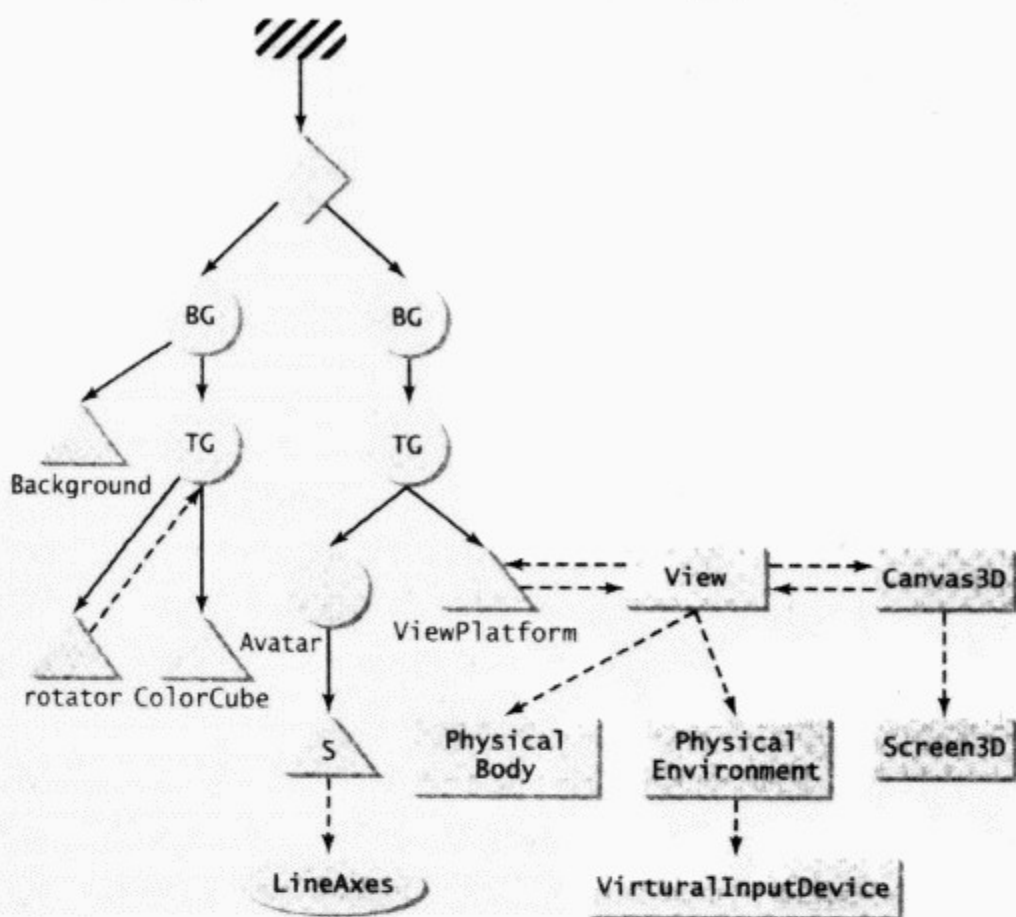


图8-16 头部跟踪场景图

主要的类和方法

- `javax.media.j3d.View` 封装了3D视图的类。
- `javax.media.j3d.ViewPlatform` 代表虚拟世界中有视图存在的叶节点类。
- `javax.media.j3d.Canvas3D` 表示Java 3D绘制表面的AWT组件。
- `javax.media.j3d.Screen3D` 封装物理显示屏的类。
- `javax.media.j3d.PhysicalBody` 封装物理实体的类。
- `javax.media.j3d.PhysicalEnvironment` 封装物理环境的类，包含传感器输入设备。
- `javax.media.j3d.InputDevice` 定义用于跟踪的输入设备的接口。
- `javax.media.j3d.Sensor` 封装跟踪设备输入值的类。
- `com.sun.j3d.util.universe.ViewerAvatar` 一个BranchGroup类，表示附加到SimpleUniverse中的Viewer的替身对象。
- `javax.media.j3d.PickShape` 表示拾取形状类族的根。
- `javax.media.j3d.BranchGroup.pickAll(PickShape)` 拾取所有与拾取形体相交的子对象的方法。在Locale中也有此方法。
- `javax.media.j3d.BranchGroup.pickAny(PickShape)` 拾取任意一个与拾取形体相交的子对象的方法。在Locale中也有此方法。
- `javax.media.j3d.BranchGroup.pickSorted(PickShape)` 拾取所有与拾取形体相交的子对象的方法。在Locale中也有此方法。
- `javax.media.j3d.BranchGroup.pickClosest(PickShape)` 拾取与拾取形体相交的最近的子对象的方法。在Locale中也有此方法。
- `com.sun.j3d.utils.picking.PickCanvas` 基于画布坐标执行拾取操作的工具类。
- `javax.media.j3d.View.setTrackingEnable(boolean)` 激活头部跟踪功能的方法。

275

关键术语

- **观察体** (view volume) 视图中可视的虚拟空间中的一部分。
- **投影矩阵** (projection matrix) 定义观察体的矩阵。
- **观察矩阵** (viewing matrix) 定义虚拟空间中视图的位置和方向的矩阵。
- **透视投影** (perspective projection) 所有的投影线都通过一个固定点的投影。
- **平行投影** (parallel projection) 所有的投影线都平行的投影。
- **vrp** (view reference point) 观察参考点, 眼睛或者视点。表示眼睛位置或者投影中心的点。
- **观察中心** (view center) 定义眼睛观察的方向的中心位置。
- **观察向上方向** (view up direction) 从观察者视角认为向上的方向。
- **FOV** (Field of view) 视域, 视图可见部分的角度。
- **前裁剪平面** (front clip plane) 可视平截体的前向平面。
- **后裁剪平面** (back clip plane) 可视平截体的后向平面。
- **长宽比** (aspect ration) 图像的高度与宽度之比。
- **兼容模式** (compatibility mode) 特殊的Java 3D视图模式, 与传统的OpenGL视图设置高度兼容。
- **6DOF** (six-degrees-of-freedom) 6自由度。提供3D位置和方向信息的设备。
- **替身** (avatar) 3D场景中观察者自身的可视表示。

本章提要

- 本章学习了3D视图的概念。视图定义了把虚拟空间场景绘制为2D图像的几何问题。视图的定义有两组特征: 与观察投影和观察体相关的属性, 以及与视图定位相关的属性。
- 投影矩阵反映了视图的投影和观察体的设置。在计算机图形学中, 通常使用两种类型的投影: 透视投影和平行投影。两者都可以用齐次坐标和投影变换表示成矩阵形式。投影矩阵定义了把指定的观察体映射到标准观察体 (例如一个中心在原点的立方体) 的一个投影变换。
- 观察矩阵表示了虚拟空间中摄像机的位置和方向。通常用一个点表示其位置, 一个点表示其观察方向, 以及对于摄像机是向上的方向来指定。观察矩阵定义了把摄像机从指定点位置映射到标准位置的变换。
- Java 3D提供了两个不同的视图模式: 兼容模式和标准的非兼容模式。前者类似于OpenGL的低层API, 提供了一种简单的方法, 用于设置基于静态摄像机的视图。标准模式提供了更为强大的功能, 比如可以支持头戴式摄像机和头部跟踪。
- 拾取是观察过程的部分逆过程。拾取指的是基于指定的拾取形体来选择虚拟世界中的对象。拾取的一个典型应用是让用户在绘制图像中选择物体。
- Java 3D在场景图中对拾取操作提供基本的支持, 这是通过拾取形体类、Locale和BranchGroup类的拾取方法, 以及表示拾取结果的BranchGroupPath。Java 3D功能类提供了方便的类, 用于简化常见的拾取操作, 比如在画布中通过鼠标事件来进行拾取。
- Java 3D视图模型集成了头部跟踪。当头部跟踪被激活之后, 可根据输入设备的头部位置信息自动计算适当的观察矩阵。

276

复习题

- 8.1 已知观察平截体的视域为 $\pi/4$, 水平边界上的两点为 $(-1, 0, 0)$ 和 $(1, 0, 0)$, 请给出视点的坐标值 (这正是由setNomialViewTransform执行的变换)。
- 8.2 计算沿着 $(1, 0, 1)$ 方向向xy平面进行平行投影的投影矩阵。
- 8.3 vrp位于原点, 投影平面垂直于z轴, 中心位于 $(0, 0, 1)$, 计算此透视投影矩阵。

8.4 找出由下面的方法定义的观察体的投影矩阵:

```
frustum(-3, 3, -2, 2, 1, 10)
```

8.5 给出由以下方法指定的观察体的投影矩阵:

```
frustum(-2, 2, -1, 1, 2, 4)
```

8.6 给出由下列方法指定的观察体的投影矩阵:

277

```
perspective(Math.PI/3, 1.5, 1, 10)
```

8.7 找出由如下方法定义的观察体的投影矩阵:

```
ortho(-3, 3, -2, 2, 1, 10)
```

8.8 找出由下述方法指定的观察体的投影矩阵:

```
ortho(-2, 2, -1, 1, 2, 4)
```

8.9 找出将视图上方改变到(1,1,0)的观察矩阵。

8.10 找出由如下方法调用定义的观察矩阵:

```
Point3d eye = new Point3d(0, 0, 0);  
Point3d look = new Point3d(0, 0, 1);  
Vector3d up = new Vector3d(0, -1, 0);  
LookAt(eye, look, up);
```

8.11 找出由以下方法调用定义的观察矩阵:

```
Point3d eye = new Point3d(0, 0, -1);  
Point3d look = new Point3d(0, 0, 0);  
Vector3d up = new Vector3d(1, 0, 0);  
lookAt(eye, look, up);
```

8.12 可以将前裁剪平面的距离设置为0吗?

编程练习

8.1 编写一个Java 3D程序,用SimpleUniverse对象显示旋转的ColorCube对象(彩色立方体)。修改视图平台变换,实现将ViewPlatform移动到(0,0,3)的功能。

8.2 修改程序清单8-1中的程序,在视图中使用平行投影。

8.3 用SimpleUniverse对象设置一个视图,并显示一个球体。将球体向靠近视图的方向移动,直到部分球体被前裁剪平面截去为止。

8.4 编写一个程序,用于测试由类Transform3D中的方法lookAt所生成的矩阵。允许用户输入方法lookAt的参数,并且输出结果矩阵。

8.5 编写一个程序,用于测试由类Transform3D中的方法perspective生成的矩阵。允许用户输入perspective方法的参数,并且输出结果矩阵。

8.6 编写一个程序,用于测试由类Transform3D中的方法frustum生成的矩阵。允许用户输入该方法所用到的参数,并且输出结果矩阵。

8.7 编写一个程序,检查由类Transform3D中的方法ortho生成的矩阵。允许用户输入该方法要用到的参数,并且输出结果矩阵。

8.8 编写一个Java 3D程序,建立一个兼容模式的视图。在程序中实现视域为 $\pi/3$ 的透视投影,让视图过点(0,0,1)并俯视z轴。程序的显示内容为第7章定义的Axes对象。

8.9 编写一个程序,建立两个不同的视图。其中,一个视图位于(0,0,2),以y轴为视图上方,注视方向为z轴负方向。另一个视图位于(0,0,-2),以y轴负方向为视图上方,注视着z轴正方向。在 origin 附近放置一个旋转的3D文本。

- 8.10 编写一个程序，建立四个视域不同的视图。所有的视图都位于 $(0, 0, 2)$ ，以 y 轴为视图上方，顺着 z 轴负方向看，视图的视域分别为 $\pi/8$ ， $\pi/4$ ， $\pi/2$ 和 $3\pi/4$ 。在场景中放置一个3D文本和一个Axes（见第7章）对象。 278
- 8.11 构建一个场景，内放一个球体和一个圆锥体，两个形体分别附加在一个TransformGroup对象上，用PickCanvas实现一个拾取操作。将通过鼠标点击选中的对象，旋转 $\pi/4$ 以后显示出来。
- 8.12 构建一个场景，内放两个球体，用PickCanvas实现一个拾取操作。将通过鼠标点击拾取的球体，变换成一个立方体后显示出来（提示：可以使用Switch节点）。
- 8.13 修改程序清单8-5，建立一个头戴式的头部跟踪视图。 279

第9章 光照与纹理

学习目标

- 理解光照模型。
- 熟悉点光源、方向光源和聚光光源。
- 认识材质属性。
- 理解Java 3D着色模型。
- 使用光源、表面法线和材质创建光照场景。
- 构造和应用雾节点(fog nodes)来进行大气衰减和景深效果处理。
- 使用2D纹理映射和纹理立体映射。
- 了解纹理坐标生成。

9.1 引言

几何属性定义了可视对象的形状和大小，绘制的图形还包括其他一些重要的属性，比如色彩、纹理以及其他的外观细节。在图形绘制中，计算量最大的任务通常与实现真实感的高质量可视对象的外观有关。

有多种不同方法来描述物体外观，它们的绘制质量和计算开销也不同。绘制可视对象的最简单的几种方法有：

- 仅仅绘制对象的顶点。
- 仅仅绘制对象的轮廓（线框模式）。
- 对象的整个表面使用单一的颜色。
- 对象正面使用单色（flat colors）。

简单的着色方法相对来说比较容易实现。但是，这些方法所产生的效果不能与人眼看到的或者照相机拍摄到的真实图像相比。

为了绘制具有真实感的3D图形，我们需要更为复杂的光照模型和明暗模型。光照模型和纹理模型都是增强绘制图形真实感的计算机图形学工具。

人肉眼所看到的真实图像，是由场景中可视对象发射或反射出来的光形成的。在自然环境中，光线的传播和相互作用是非常复杂的，不同光源的特征各不相同。各种对象反射和折射光的特性也不一样，它们反射的光线可能在场景中通过复杂的路径。

在计算机中，要为如此复杂的系统建立精确的模型，一般是不可行的。因此，为了实现一个可用的图形系统，必须使用各种不同的近似方法。计算机图形系统一般采用简化的光照模型，来绘制具有一定真实感的场景。phong光照模型受到了广泛的应用，为了获得有效光照明计算，该模型包含了光源特征、几何体结构以及材质属性等三方面的信息。

用来改善绘制效果的技术一般还包括大气衰减和纹理映射。在大气衰减和景深效果处理中，使用人造雾使得远方的对象变模糊，该方法生成一个雾化外观。纹理映射技术将图像映射到可视对象的表面，它的优点是不需要复杂的模型，就可以生成大量的表面细节。

9.2 光源

在虚拟世界里，光源提供了图形对象的光照来源。光源具有颜色、位置、方向以及其他一些与特定光源类型相关的属性。在图形系统里，有四种典型的光源：环境光源、方向光源、点光源和聚光光源。

环境光 (ambient light) 在各方向和各位置都是均一的。在真实世界场景中，物体之间存在大量强度较弱的相互反射，通常，环境光源就是这些反射光的简化表示 (如图9-1所示)。

282

方向光(directional light)有一个固定的方向，所有光线都沿该方向传播。尽管可以认为方向光是从无限远处发射出来的，但是方向光没有具体固定的位置。例如太阳这种位于远处的光源，就是一个很好的方向光模型 (如图9-2所示)。

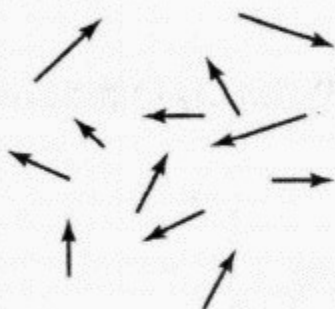


图9-1 环境光用较弱的随机反射来表示

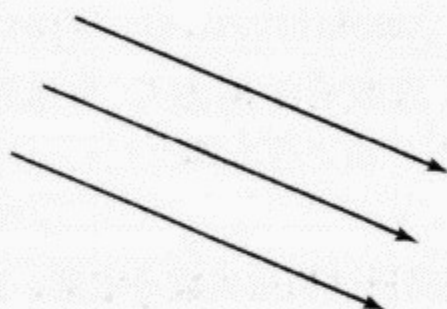


图9-2 方向光发出平行的光线

在虚拟世界中，点光源 (point lights) 是有位置的，它向各个方向发射光线(如图9-3所示)。点光源一般带有强度衰减效果，即随着距离增加光线强度不断减弱，通常将衰减表示为从0到1之间的一个系数 (与距离相关的函数) 乘以光线的原始强度。

聚光光源 (spotlight) 与点光源相似，但是其光线传播的方向受到限制，聚光光源仅向一个固定的圆锥形区域发射光线 (如图9-4所示)。除了与到光源的距离相关的衰减之外，聚光光源还可以定义当光线偏离圆锥的中心轴方向时的衰减。

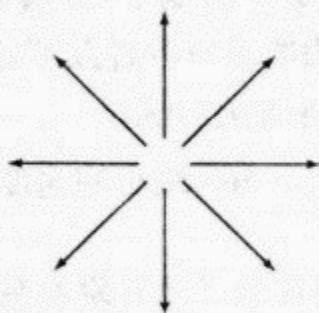


图9-3 点光源有特定的位置并向各方向发射光线

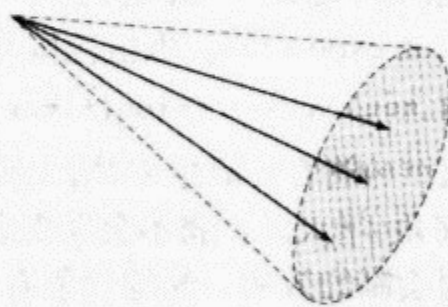


图9-4 聚光光源向一个圆锥形区域内发射光线

283

Java 3D为上述不同类型的光源提供了一个Light类族，Light类的层次结构如图9-5所示。

下面给出了Light类的一些构造函数：

```
// 环境光源的类构造函数
AmbientLight()
AmbientLight(boolean lightOn, Color3f color)
AmbientLight(Color3f color)
// 方向光源的类构造函数
DirectionalLight()
DirectionalLight(boolean lightOn, Color3f color, Vector3f direction)
DirectionalLight(Color3f color, Vector3f direction)
// 点光源的类构造函数
PointLight()
```

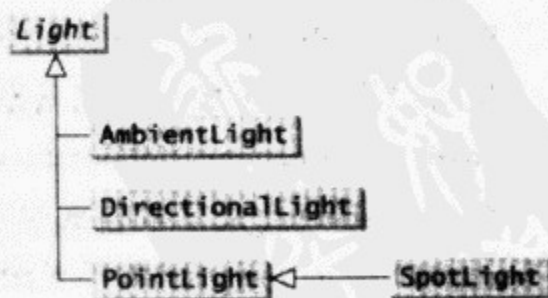


图9-5 光源Light类


```

PointLight(boolean lightOn, Color3f color,
Point3f position, Point3f attenuation)
PointLight(Color3f color, Point3f position, Point3f attenuation)
// 聚光光源的类构造函数
SpotLight()
SpotLight(boolean lightOn, Color3f color,
Point3f position, Point3f attenuation,
Vector3f direction, float spreadAngle, float concentration)
SpotLight(Color3f color, Point3f position, Point3f attenuation,
Vector3f direction, float spreadAngle, float concentration)

```

光源可以被打开或关闭。除了设置构造函数中的状态标记之外，还可以通过调用下述方法来设置打开/关闭的状态：

```
void setEnable(boolean state) //光源状态设置函数
```

假设到光源的距离为 d ，可以用一个二次模型来定义PointLight和SpotLight的衰减(attenuation)。定义形式如下：

$$A(d) = 1/(a_0 + a_1d + a_2d^2)$$

衰减可以通过构造函数来定义，或者利用下述方法进行设置：

```
void setAttenuation(Point3f attenuation) //光源衰减设置函数
```

用Point3f参数来指定衰减系数 (a_0, a_1, a_2) 。例如，系数 $(1,0,0)$ 意味着光线没有衰减， $A(d) = 1$ 。系数为 $(1,1,0)$ 时，衰减函数可以表示为 $A(d) = 1/(1+d)$ ，因此在离光源的距离为1米时，衰减函数的值为0.5，距离为4米时，衰减为0.2。

聚光光源的光线偏离其中心轴方向时，它的衰减可以用其发散角余弦值的指数函数来定义：

$$A(\theta) = \cos^n \theta$$

当夹角为0度时，函数值为1，因此在中心轴方向没有衰减。参数 n 是聚集指数(concentration exponent)，取值范围是从0到128，它可以通过构造函数或者以下函数来设置：

284 `void setConcentration(float concentration) //聚集指数设置函数`

函数值 $A(\theta)$ 随着发散角 θ 的增大而减小。如果聚集指数较大， $A(\theta)$ 减小的速度会更快。如果 $n = 0$ ，当夹角 θ 增大时，将不会发生衰减。

考虑到绘制的效率，光源对象有一个相关的影响范围。只有当可视对象处在光源的照射范围内时，才能受到光源Light类对象的作用。每创建一个光源，都有必要设置一个合适的影响范围，从而可以避免不必要的繁琐绘制过程，下述方法用来设置作用区域：

```
void setInfluencingBounds(Bound bounds)
```

也可以用一个区域叶节点来指定对应的影响范围：

```
void setInfluencingBoundingLeaf(BoundingLeaf bounds)
```

程序清单9-1演示了不同类型光源的照射效果。程序显示一个受到不同光源照射的球体（如图9-6所示），可以用一组关于光源类型的检查框来单独打开或关闭每种光源。

程序清单9-1 TestLights.java

```

1 package chapter9;
2
3 import javax.vecmath.*;
4 import java.awt.*;

```



```
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义TestLights类, 继承自Applet类, 实现ItemListener接口
12 public class TestLights extends Applet implements ItemListener {
13     public static void main(String[] args) {
14         new MainFrame(new TestLights(), 640, 480); //创建程序主窗口并设置大小
15     }
16
17     AmbientLight aLight;
18     DirectionalLight dLight;
19     PointLight pLight;
20     SpotLight sLight;
21     SpotLight sLight2;
22     //重写初始化方法
23     public void init() {
24         //创建画布
25         GraphicsConfiguration gc =
26             SimpleUniverse.getPreferredConfiguration();
27         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D类对象来显示3D场景
28         setLayout(new BorderLayout());
29         add(cv, BorderLayout.CENTER);
30
31         Panel menu = new Panel(); //创建用来放置光源类型复选框的面板
32         menu.setLayout(new GridLayout(1,4)); //设置该面板的布局
33         add(menu, BorderLayout.SOUTH);
34         Checkbox mi = new Checkbox("Ambient", true); //创建复选框对象
35         menu.add(mi);
36         mi.addItemListener(this); //注册事件侦听器
37         mi = new Checkbox("Directional", true);
38         menu.add(mi);
39         mi.addItemListener(this);
40         mi = new Checkbox("Point", true);
41         menu.add(mi);
42         mi.addItemListener(this);
43         mi = new Checkbox("Spot", true);
44         menu.add(mi);
45         mi.addItemListener(this);
46
47         SimpleUniverse su = new SimpleUniverse(cv, 2); //创建设置SimpleUniverse对象
48         su.getViewingPlatform().setNominalViewingTransform();
49         BranchGroup bg = createSceneGraph(su.getViewingPlatform().
50             getMultiTransformGroup().getTransformGroup(0)); //创建一个场景分支
51         bg.compile();
52         su.addBranchGraph(bg);
53     }
54
55     Appearance ap;
56     private BranchGroup createSceneGraph(TransformGroup vtg) {
57         BranchGroup root = new BranchGroup(); //创建分支的根节点
58         //创建球体
```

```

59     ap = new Appearance();//为该球体创建外观对象
60     ap.setMaterial(new Material());//设置材质属性
61     Sphere shape = new Sphere
62         (0.5f, Sphere.GENERATE_NORMALS, 150, ap); //创建球体*/
63     root.addChild(shape);//添加球体到场景子图中
64     //旋转
65     Alpha alpha = new Alpha(-1, 4000);
66     RotationInterpolator rotator = new RotationInterpolator
67         (alpha, vtg);
68     BoundingSphere bounds = new BoundingSphere();//创建一个行为区域球体
69     bounds.setRadius(2);//设置该区域球体的半径为2
70     rotator.setSchedulingBounds(bounds);
71     root.addChild(rotator);
72     //添加背景与光源
73     Background background = new Background(0.5f, 0.5f, 0.5f);
74     background.setApplicationBounds(bounds);
75     root.addChild(background);
76     aLight = new AmbientLight(true, new Color3f(Color.red));//添加红色环境光源
77     aLight.setInfluencingBounds(bounds);//设置该环境光源的影响范围
78     aLight.setCapability(Light.ALLOW_STATE_WRITE);
79     root.addChild(aLight);
80     dLight = new DirectionalLight(
81         new Color3f(Color.green), new Vector3f(0f,1f,0f)); //创建绿色平行光源
82     dLight.setCapability(Light.ALLOW_STATE_WRITE);
83     dLight.setInfluencingBounds(bounds);//设置该光源的影响范围
84     root.addChild(dLight);//将该光源添加到场景子图中
85     pLight = new PointLight(
86         new Color3f(Color.white), new Point3f(-0.7f,0.7f,0f),
87         new Point3f(1f,0f,0f)); //创建白色点光源
88     pLight.setCapability(Light.ALLOW_STATE_WRITE);
89     pLight.setInfluencingBounds(bounds); //设置该光源的照射范围
90     root.addChild(pLight);//将该光源添加到场景子图中
91     sLight = new SpotLight(
92         new Color3f(Color.blue), new Point3f(0.7f,0.7f,0.7f),
93         new Point3f(1f,0f,0f),
94         new Vector3f(-0.7f,-0.7f,-0.7f), (float)(Math.PI/6.0), 0f);// 创建蓝色聚光光源
95     sLight.setCapability(Light.ALLOW_STATE_WRITE);
96     sLight.setInfluencingBounds(bounds);//设置该光源的照射范围
97     root.addChild(sLight);//将该光源添加到场景子图中
98     sLight2 = new SpotLight(
99         new Color3f(Color.orange), new Point3f(0.7f,0.7f,-0.7f),
100         new Point3f(1f,0f,0f),
101         new Vector3f(-0.7f,-0.7f,0.7f), (float)(Math.PI/12.0), 128f);//创建橙色聚光光源
102     sLight2.setCapability(Light.ALLOW_STATE_WRITE);
103     sLight2.setInfluencingBounds(bounds);//设置该光源的照射范围
104     root.addChild(sLight2);//将该光源添加到场景子图中
105     return root;
106 }
107 事件响应函数
108 public void itemStateChanged(ItemEvent itemEvent) {
109     Checkbox cmi = (Checkbox)itemEvent.getSource();
110     String label = cmi.getLabel();
111     boolean state = cmi.getState();
112     if("Ambient".equals(label)) {

```



```
113     aLight.setEnabled(state);
114 } else if ("Directional".equals(label)) {
115     dLight.setEnabled(state);
116 } else if ("Point".equals(label)) {
117     pLight.setEnabled(state);
118 } else if ("Spot".equals(label)) {
119     sLight.setEnabled(state);
120     sLight2.setEnabled(state);
121 }
122 cmi.setState(state);
123 }
124 }
```

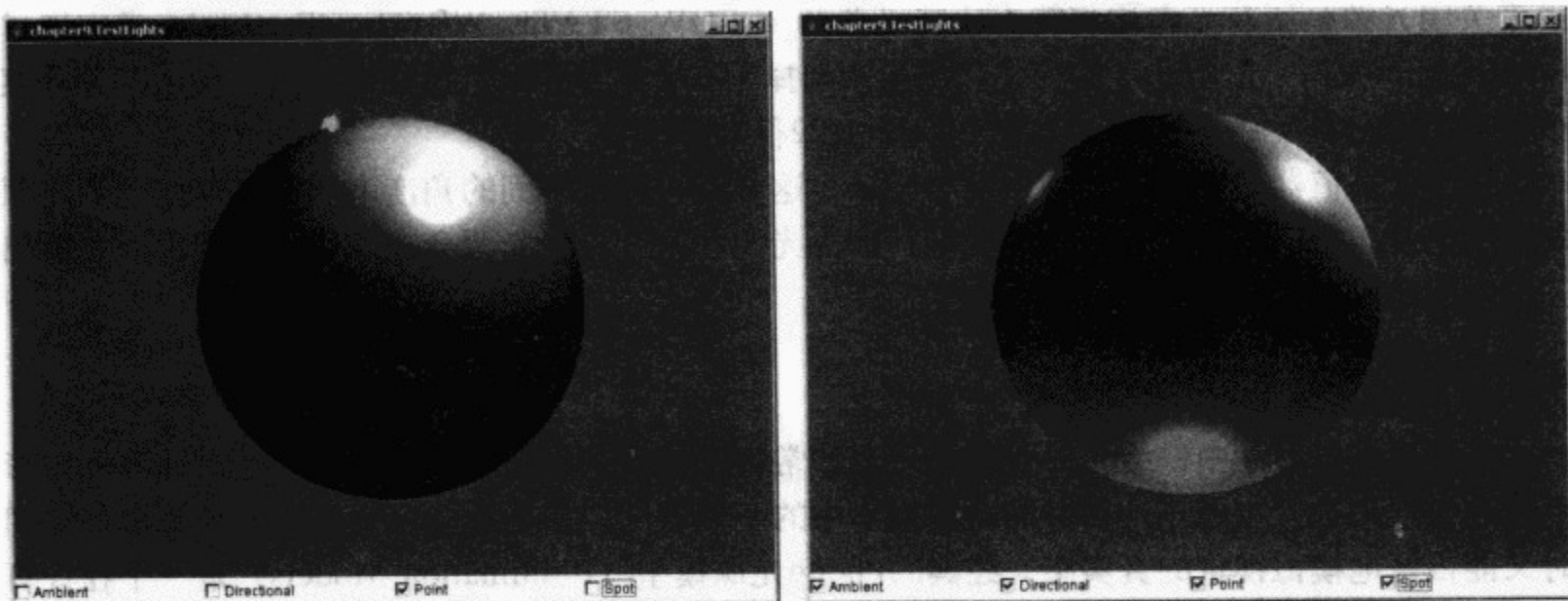


图9-6 不同类型光源的效果图。左图:只有一个点光源。右图:四种类型的光源都有

该程序的场景图如图9-7所示。我们创建了四个不同类型的光源来照射该场景(一个环境光源、一个方向光源、一个点光源以及两个聚光光源)。环境光源是红色的,方向光源是绿色的,光线指向(0,1,0),因此其光线是向正上方发射的。点光源位于(-0.7,0.7,0),发白光并且没有衰减。第一个聚光光源发蓝光,光源位置是(0.7,0.7,0.7),并且其衰减与距离无关,该光源以大小为 $\pi/6$ 的发散角向方位(-0.7,-0.7,-0.7)发射光线。聚集指数设为0,因此衰减与发散角大

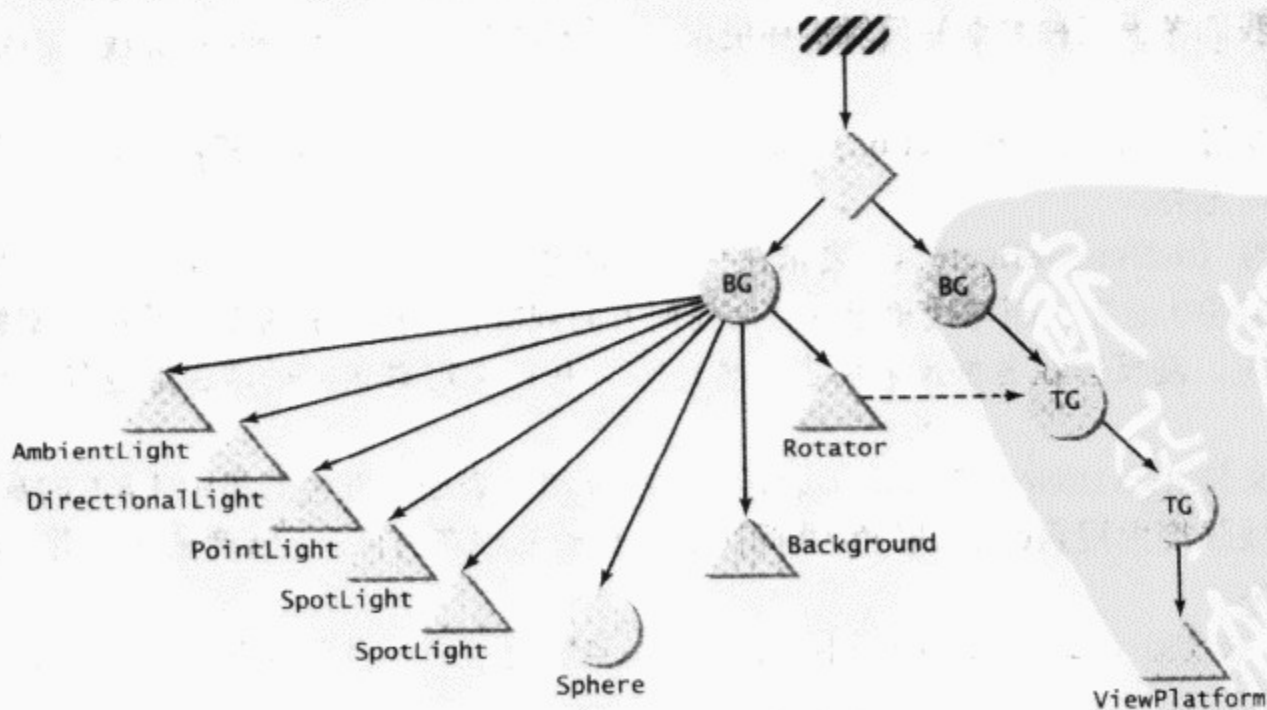


图9-7 该实例的场景图

小无关。第二个聚光光源发橘黄色的光，光源位置是 $(0.7, 0.7, -0.7)$ ，而且衰减与距离无关。该光源向方位 $(-0.7, -0.7, 0.7)$ 发射光线，发散角是 $\pi/12$ 。聚集指数设为128，因此光线沿着主光轴高度聚集。所有光源都用同一个Bounds类对象来设置它们的影响范围，该球体作用范围边界的球面半径为2，可以包含所有对象。添加一个灰色背景对象，它也是使用同一个球体作用范围边界。

287

该场景中的可视对象是一个半径为0.5的球体（第61行），我们将该球体由150个单元拼接而成，目的是为了得到一个高质量的球体，球体以坐标原点 $(0, 0, 0)$ 为中心。为了支持光照，该球生成了曲面法线，并且在它的Appearance对象中包含一个Material对象。

在窗口的底部，创建了Checkbox类型的四个检查框，分别是Ambient、Directional、Point和Spot。点击这些检查框，就可以打开或者关闭相应的光源。为了能在场景图中实时地选择打开或者关闭光源，所有光源均将能力比特设置为ALLOW_STATE_WRITE。TestLights类实现了ItemListener接口来监听选项的变化，当用户选择了一个菜单项时，该光源就会被激活，并且菜单项上的标记就反映了光源的状态（第109~122行）。

用一个插值器对象来驱动视图发生旋转，这样就可以从不同的角度来展现该球体受到光源照射的情况。方向光源照射球体的底端，而点光源和两个聚光光源会在球体的上半部分照射出三个不同的光斑。

9.3 光照模型

在计算机图形学中，给光反射建立一个非常精确的模型将会极其复杂，并且对计算机图形系统的实现来说，也是不切实际的。图形学中的一般做法，就是建立一个高效的计算模型，同时又能很好地模拟视觉的真实感。经典的Phong光照模型（Illumination Models）是一个在计算效率和绘制质量两者之间试图取得平衡的模型，它在计算机图行学中得到了广泛的使用。

图9-8显示的是光线照射在表面上一点的情形。该点的表面法线N是一条与这个点所在的切平面相垂直的直线，L是入射光线的方向，L与法线N的夹角为 θ 。R是L的反射方向，它与法线N的夹角也为 θ 。V是视角方向，我们将它与反射方向R之间的夹角定义为 α 。

288

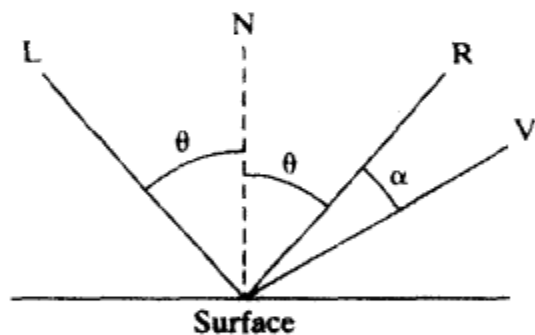


图9-8 光线反射的几何体

通常我们考虑三种类型的反射：环境反射、漫反射以及镜面反射。

环境反射（ambient reflection）与环境光源相对应，它在各方向都是一致的，且和物体表面法线方向无关。

漫反射（deffuse reflection）表示在一个不是绝对光滑的表面上发生的大量小反射。反射光强由入射角 θ 决定，而与视角无关。从任何方向观察，反射光强都是一样的。对绝大多数可视对象来说，漫反射在各种反射中占主要部分，并且决定着我们会形成什么样的对象颜色概念。

镜面反射（specular reflection）表示在一个光滑的表面上发生的集中于R方向周围的反射，其反射光线强度由视角 α 与入射角 θ 共同决定。反射光线在方向R上强度最大，并且随着视角 α 增大而减小。

在Phong模型（Phong model）中，表面上一点的光线强度可以由下面的光照公式给出：

$$I = I_a k_a + I_p k_d \cos \theta + I_p k_s \cos^n \alpha$$

在公式中, I_a 和 I_p 分别表示环境光源和点状光源发出的光线强度, 在 I_p 中光线的衰减已经得到考虑。系数 k_a 、 k_d 与 k_s 分别是对应于环境反射、漫反射与镜面反射的反射系数, 这些系数由对象反射表面的材质属性来决定。 n 是镜面反射指数, 它也是由材质决定, 指数越大, 高光区域越集中。

上述光照公式适用于任何波长的光波。因此, 对于不同颜色的光, 系数 k_a 、 k_d 与 k_s 可能不相同。当然, 对波长连续的光谱, 采用该光照公式计算是不现实的, 一般做法就是选用诸如RGB模型之类的一种颜色模型, 然后针对该颜色模型中的每一个颜色分量 (RGB模型中的红、绿、蓝) 应用上述光照公式, 来计算各个颜色分量的光照强度。

在有多个光源的情况下, 将来自每一个光源的光强相加, 就可以得到总的光线强度。

Phong模型是一种局部光照模型, 它单独绘制每一个可视对象, 而不考虑它们之间的相互影响。例如, 它不会考虑其他物体产生的反射和阴影。像辐射度 (radiosity) 和光线跟踪 (ray tracing) 这些全局模型和技术, 可以解决这些问题, 但是这些模型比Phong模型付出的计算代价要高很多。

从光照模型公式很容易看到, 可视对象的外表不仅与决定如何对光照做出反应的材质属性有关, 还与物体的几何特征和光源有关。

材质属性主要由反射系数(reflection coefficients) k_a 、 k_d 与 k_s 来表示, 这些系数取决于颜色光的波长。当使用颜色模型时, 对该颜色模型的每个分量都要指定反射系数。例如, 在RGB模型中, 可以通过 (0.9, 0.2, 0.0) 三个数来确定漫反射系数 k_d 。这表示在漫射中, 这种材质将主要反射红光和少量绿光而不反射蓝光, 因此可视对象看起来将主要呈红色。镜面光滑指数 (shininess exponent) n 也是一种材质属性, 但是通常将它指定为一个与光颜色无关的固定常数。

289

9.4 Java 3D 光照模型

Java 3D支持Phong光照模型。当一个可视对象受到光照时, 就可以依据该模型来进行绘制。Java 3D采用RGB颜色模型来绘制物体, 光照公式用于颜色空间的三个颜色分量。

Material节点组件类表示可视对象的材质属性。在这个类里面, 环境反射、漫反射和镜面反射的系数用RGB颜色模型来表示, 镜面反射指数 n 与Material类中的光滑指数是一致的。Material类还包含一个发光系数, 该发光系数定义除去来自外部光源外, 对象自身发射出的光。

下面列出了Material类中, 用于设置各种参数的部分函数:

```
void setAmbientColor(Color3f color)
void setAmbientColor(float r, float g, float b) // 设置材质的环境反射颜色
void setDiffuseColor(Color3f color)
void setDiffuseColor(float r, float g, float b)
void setDiffuseColor(float r, float g, float b, float a) //设置材质的漫反射颜色
void setSpecularColor(Color3f color)
void setSpecularColor(float r, float g, float b) //设置材质的高光反射颜色
void setEmissiveColor(Color3f color)
void setEmissiveColor(float r, float g, float b) // 设置材质的发光颜色
```

对一个可视对象进行着色, 有三种不同的方式: 在其Geometry对象中定义顶点颜色, 在其Appearance对象中定义ColoringAttributes属性, 以及采用光照模型计算颜色值。对一个特定物体进行着色, 可以按以下步骤来选择具体的着色方法:

- 1) 如果对对象的几何体指定了顶点颜色, 那么就可以用这些顶点颜色对对象进行着色。
- 2) 如果为一个对象定义了一个Material节点组件, 那么意味着它接受光照。因而, 可以通过光照模型对该对象进行着色。

- 3) 如果物体的外观中定义了ColoringAttributes属性, 那么就可以用该颜色对对象进行着色。
- 4) 在其他情况下, 对象都设置为白色。

每个对象的着色方式是单独选取的, 这就允许在同一场景中既可以出现接受光照的对象, 也可以含有不接受光照的对象。例如, 下面的程序片段用不同的着色方式创建了四个对象:

```
// 在几何体中指定顶点颜色
TriangleArray geom = new TriangleArray(3,
TriangleArray.COORDINATES|TriangleArray.COLOR_3); // 创建一个三角片面geom
Point3f[] coords = {new Point3f(1,0,0),
new Point3f(0,1,0), new Point3f(0,0,1)}; // 创建一个顶点数组coords
geom.setCoordinates(coords); // 设置该三角片面的顶点坐标
Color3f[] colors = {new Color3f(1,0,0),
new Color3f(0,1,0), new Color3f(0,0,1)}; // 创建一个颜色数组
geom.setColors(colors); // 用该颜色数组来设置顶点颜色
Shape3D shape1=new Shape3D(geom); // 利用该三角片面来构造一个Shape3D可视对象shape1

// 接受光照的对象
TriangleArray geom2 = new TriangleArray(3,
TriangleArray.COORDINATES|TriangleArray.NORMAL); // 创建一个三角片面geom2
geom2.setCoordinates(coords); // 用顶点数组coords设置该三角片面的顶点坐标
Appearance appear = new Appearance(); // 创建一个外观组件
Material material = new Material(); // 创建一个材质属性
appear.setMaterial(material); // 将已创建的材质属性设置为该外观的材质属性
Shape3D shape2 = new Shape3D(geom2,appear); // 用该三角片面和外观组件来构造一个Shape3D可
视对象shape2

// 外观颜色分布
Appearance appear3 = new Appearance(); // 创建一个外观组件
ColoringAttributes coloring = new ColoringAttributes(
new Color3f(1f,0,0),ColoringAttributes.SHADE_FLAT); // 创建一个颜色属性
appear3.setColoringAttributes(coloring); // 设置该外观的颜色属性为coloring
Shape3D shape3 = new Shape3D(geom2,appear3); // 用该三角片面和外观组件来构造一个
Shape3D可视对象shape2

// 没有指定颜色
Shape3D shape4 = new Shape3D(geom2);
```

类Shape3D的对象shape1有一个包含顶点颜色定义的几何体。因此, 可以用所定义的顶点颜色进行绘制。对象shape2在它的几何体中没有定义顶点颜色, 但是, 在每个顶点位置定义了法线。对象shape2中的外观属性包含一个Material类的对象, 因而shape2可受到光照, 并且其着色将从光照模型中获得。对象shape3与对象shape2使用的几何体是相同的, 没有定义顶点颜色, 但是也不受光照。shape3的外观属性包含一个ColoringAttributes类的对象, 该对象指定颜色为红色, 因此shape3将会着色为红色, 对象shape4没有定义顶点颜色, 也不包含Material类和ColoringAttributes类的对象, 它将会着色为白色。

因此, 在Java 3D程序中, 要使得可视对象能受到光照, 需要执行以下基本步骤:

- 在场景图中放置光源。
- 给出几何体的表面法线, 同时不要设置顶点颜色。
- 为该对象的外观设置材质属性。

可用以下程序片段来说明上述步骤:

```
// 添加一个光源
Light light = new PointLight(); // 创建一个点光源
```



```

light.setInfluencingBounds(new BoundingSphere());
root.addChild(light); // 将光源添加到场景图中

// 构造一个具有材质属性的外观
Appearance appearance = new Appearance(); // 创建一个外观组件
Material material = new Material();
appearance.setMaterial(material); // 设置该外观的材质属性

// 添加一个含有法线的Box类可视对象box
Box box = new Box(1f, 1f, 1f, Box.GENERATE_NORMALS, appearance); // 用该外观
创建一个带有法线的Box类可视对象box, 其长、宽、高均为1
root.add(box); // 将该可视对象添加到场景图中

```

程序清单9-2给出了在不同光照参数设置下的物体着色效果, 这是基于八个具有不同材质属性的球体, 接受同一点光源照射而得出的。

程序清单9-2 Lighting.java

```

1 package chapter9;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义Lighting类, 继承自Applet类
12 public class Lighting extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new Lighting(), 640, 480); //创建主窗口并设置大小
15     }
16     //重写初始化方法
17     public void init() {
18         //创建画布
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
21         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D类对象
22         setLayout(new BorderLayout()); //设置布局管理器
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph(); //构造场景图
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse对象
27         su.getViewingPlatform().setNominalViewingTransform();
28         su.addBranchGraph(bg);
29     }
30     //生成BranchGroup的私有方法, 构造场景图
31     private BranchGroup createSceneGraph() {
32         BranchGroup root = new BranchGroup();
33         //创建几何体
34         Vector3f pos = new Vector3f(0.5f, 0f, 0f);
35         Transform3D rotation = new Transform3D(); //创建一个Transform3D对象
36         rotation.rotZ(Math.PI/4); //设置图形Z轴方向沿逆时针旋转, 角度为45度
37         for (int i = 0; i < 8; i++) { //依次创建八个球体, 并将球体添加到指定的位置

```

```

38     Node shape = createShape
39         (pos, 0.2f, .5f, (float)Math.pow(1.7,i));
40     root.addChild(shape);
41     rotation.transform(pos);
42 }
43 //添加光源
44 BoundingSphere bounds = new BoundingSphere();
45 PointLight pLight = new PointLight(new Color3f(Color.white),
46     new Point3f(0f,0f,1f/(float)Math.tan(Math.PI/8)),
47     new Point3f(1f,0f,0f)); //创建白色点光源, 给出位置及衰减系数
48 pLight.setInfluencingBounds(bounds); //设置点光源的区域范围
49 root.addChild(pLight);
50 return root;
51 }
52 //生成Node对象的私有方法, 创建几何图形
53 private Node createShape(Vector3f pos,
54     float size, float spec, float shine) {
55     Material mat = new Material();
56     mat.setDiffuseColor(0.5f,0.5f,1f); //设置材质的漫反射颜色
57     mat.setSpecularColor(spec, spec, spec); //设置材质的高光反射颜色
58     mat.setShininess(shine); //设置材质的Shininess指数
59     Appearance ap = new Appearance(); //创建外观对象
60     ap.setMaterial(mat);
61     Sphere shape = new Sphere(size, Sphere.GENERATE_NORMALS, 50, ap);
62     Transform3D tr = new Transform3D();
63     tr.setTranslation(pos);
64     TransformGroup tg = new TransformGroup(tr);
65     tg.addChild(shape);
66     return tg;
67 }
68 }

```

该程序所绘制的场景如图9-9所示, 图9-10是该程序的场景图。光滑指数不同的八个球体环绕z轴放置, 绕z轴每隔 $\pi/4$ 就计算出一个球体的位置。第一个球体位于坐标 (0.5,0,0) 处, 使用一个Transform3D类对象, 在该位置向量上做旋转。

292

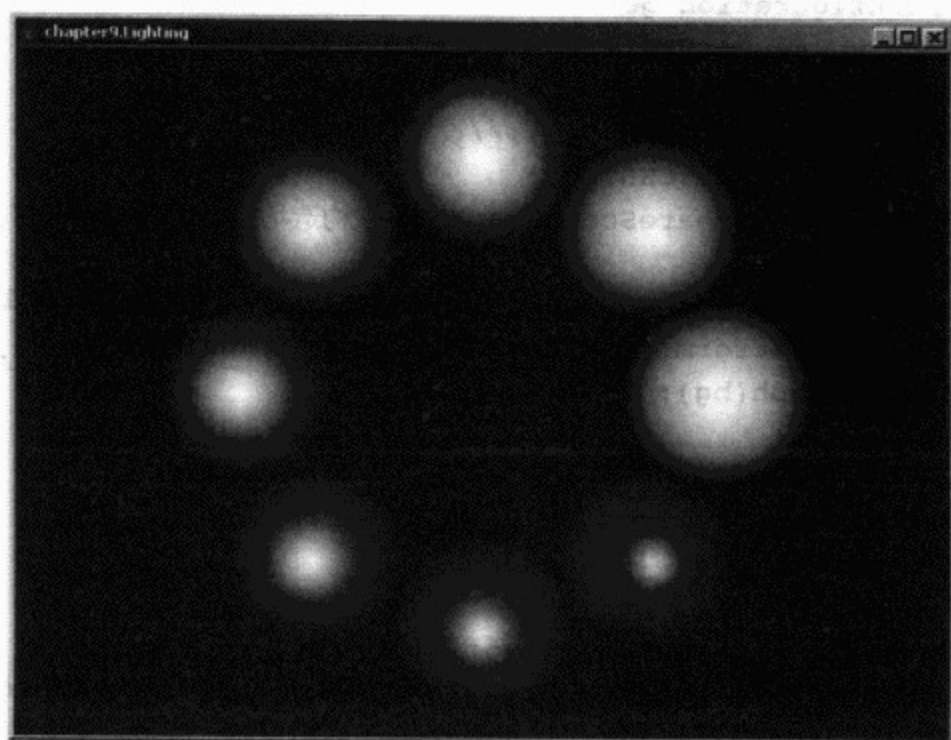


图9-9 材料属性

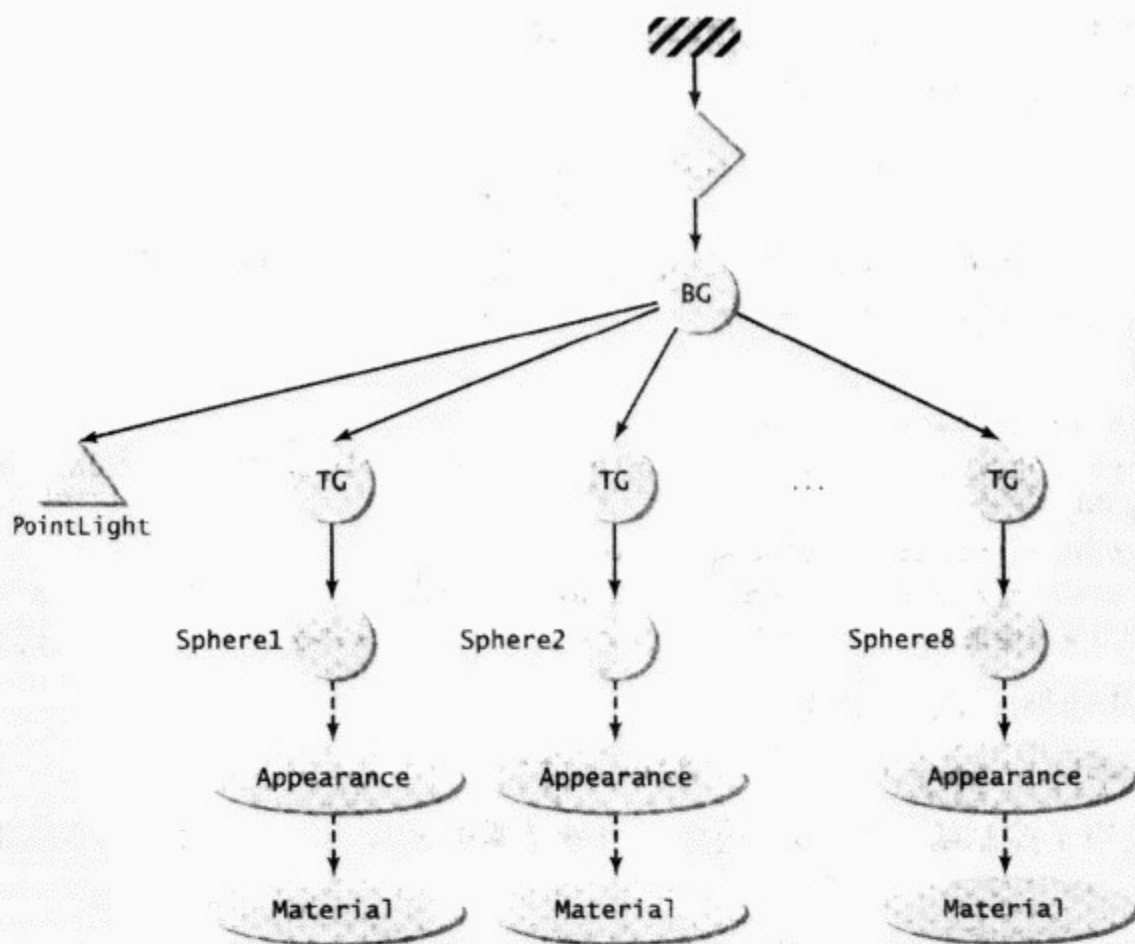


图9-10 场景图

293

方法createShape（第53行）创建了场景图的一个分支，该分支包含一个TransformGroup节点和一个球体。TransformGroup节点将球体移动到指定的位置，该球体的外观属性包含在Material类的对象中，其光滑指数被设为一个指定的值，同时将漫反射设置为一个固定的值。

为了使得光照和视图效果对于所有球体是相同的，我们将点光源设置在z轴上人眼所在的位置，这样，当光滑指数很大的时候，聚集的镜面反射仍然是可见的。八个球体的光滑指数值分别设为 1.7^i ， $i=0, 1, \dots, 7$ 。第一个球的亮度值最小，取值为1，最后一个球的亮度值最大，取值是41。

9.5 大气衰减和景深效果处理

在现实生活的图像中，对于观测者来说，近处的对象比远处的对象看起来更清晰、更明亮、光线更强。当天气有雾时，这个现象特别明显。这个现象就是大气衰减造成的，这需要对应的一种技术来提高它的真实感。在计算机图形学中，可以通过一个相对高效的方法，来模拟大气衰减所产生的效果，这就是景深效果处理（Depth Cueing）。景深效果处理将对象与背景颜色相混合，用于混合的背景颜色的比重是一个关于距离的递增函数。更一般的是，雾的颜色——该颜色用来与对象相混合——可以任意指定。

某个绘制点的最终颜色可由以下公式得出：

$$C = f \cdot C_0 + (1-f) \cdot C_f$$

其中 C_0 是对象本身的颜色， C_f 是雾的颜色，雾因子 f 是一个关于到观察者距离的递减函数。因此，当对象距离观察者越远，雾的颜色权重越大，而对象自身颜色的权重越小。此时，对象看起来将会消失在雾里。通常情况下，雾因子 f 可以看成是距离的线性函数、指数函数或高斯函数。

Java 3D提供Fog类叶节点来支持景深效果处理。在Java 3D中提供两

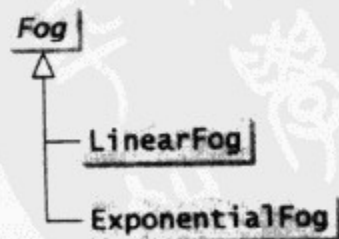


图9-11 雾类Fog的层次关系

种类型的雾，其雾因子分别是线性和指数混合函数。图9-11给出了雾的有关类。

LinearFog类定义一种雾，可以用一个线性函数进行表示。

$$f(z) = \frac{back - z}{back - front}$$

其中 z 是到观察点的距离， $front$ 和 $back$ 是表示雾范围的常数， f 的值在范围 $[0.0, 1.0]$ 之间变化。可以用下列构造函数来创建一个线性雾：

```
public LinearFog() // 默认构造函数
public LinearFog(Color3f color)
public LinearFog(Color3f color, double front, double back) // 用指定的颜色color及雾范围来构造一个线性雾类节点
public LinearFog(float r, float g, float b)
public LinearFog(float r, float g, float b, double front, double back) // 用指定的颜色及雾范围来构造一个线性雾类节点，r、g、b分别表示RGB模型中红、绿、蓝三个颜色分量
```

ExponentialFog类使用指数函数

$$f(z) = e^{-density \cdot z}$$

density参数用于控制函数 f 的递减速度，也就是雾的密度。可以采用以下构造函数来创建指数雾：

```
public ExponentialFog() // 默认构造函数，默认密度指数density=1.0
public ExponentialFog(Color3f color)
public ExponentialFog(Color3f color, float density) // color为指数雾的颜色，density为雾的密度
public ExponentialFog(float r, float g, float b)
public ExponentialFog(float r, float g, float b, float density) // r、g、b分别是雾颜色RGB模型的红、绿、蓝三个分量，density为雾的密度
```

程序清单9-3演示了如何使用雾节点，程序运行的结果是一组十二面体出现在灰色背景里面，将LinearFog节点放置在场景中。雾的颜色和背景相同，远处的对象消失在背景中，从而得到了雾环境的效果。

程序清单9-3 TestFog.java

```
1 package chapter9;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import chapter6.Dodecahedron;
10 import java.applet.*;
11 import com.sun.j3d.utils.applet.MainFrame;
12 //定义TestFog类，继承自JApplet类，演示使用雾节点
13 public class TestFog extends Applet {
14     public static void main(String[] args) {
15         new MainFrame(new TestFog(), 640, 480); // 创建主窗口并设置大小
16     }
17     //重写初始化方法
18     public void init() {
19         //创建画布
```



```
20 GraphicsConfiguration gc =
21     SimpleUniverse.getPreferredConfiguration();
22 Canvas3D cv = new Canvas3D(gc); //用gc构造Canvas3D对象
23 setLayout(new BorderLayout()); //设置布局管理器
24 add(cv, BorderLayout.CENTER);
25 BranchGroup bg = createSceneGraph(); //创建场景分支图
26 bg.compile();
27 SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
28 su.getViewingPlatform().setNominalViewingTransform();
29 su.addBranchGraph(bg);
30 }
31 //生成BranchGroup的私有方法, 创建场景分支图
32 private BranchGroup createSceneGraph() {
33     BranchGroup root = new BranchGroup(); //创建场景分支子图的根节点
34     //添加场景物体
35     for (int i = 0; i < 5; i++) { //依次创建25个场景物体, 并添加到场景分支图中
36         for (int j = 0; j < 5; j++) {
37             Vector3f pos = new Vector3f
38                 (-0.8f+0.4f*i, -0.2f+0.2f*j, -0.4f*j); //位置坐标
39             Node shape = createShape(pos); //创建物体
40             root.addChild(shape);
41         }
42     }
43     //添加光源
44     BoundingSphere bounds = new BoundingSphere
45         (new Point3d(), Double.MAX_VALUE);
46     Background background = new Background(.6f, .6f, .6f); //创建背景对象
47     background.setApplicationBounds(bounds);
48     root.addChild(background);
49     DirectionalLight dLight = new DirectionalLight
50         (new Color3f(Color.white), new Vector3f(1f, 0f, -1f)); //构造光源
51     dLight.setInfluencingBounds(bounds);
52     root.addChild(dLight);
53     //添加雾效果
54     LinearFog fog = new LinearFog(.6f, .6f, .6f, 2f, 4f); //生成线性雾
55     fog.setInfluencingBounds(bounds); //设置线性雾的行为区域
56     root.addChild(fog); //将线性雾添加到场景分支图中
57     return root;
58 }
59 //创建场景物体的私有方法
60 private Node createShape(Vector3f pos) {
61     Material mat = new Material(); //构造材质
62     Appearance ap = new Appearance(); //生成外观对象
63     ap.setMaterial(mat); //设置材质属性
64     Shape3D shape = new Dodecahedron(); //创建一个十二面体
65     shape.setAppearance(ap);
66     Transform3D tr = new Transform3D();
67     tr.setScale(0.1); //缩放
68     tr.setTranslation(pos); //平移
69     TransformGroup tg = new TransformGroup(tr);
70     tg.addChild(shape);
71     return tg;
72 }
73 }
```


图9-12给出该程序的执行结果，场景图如图9-13所示。

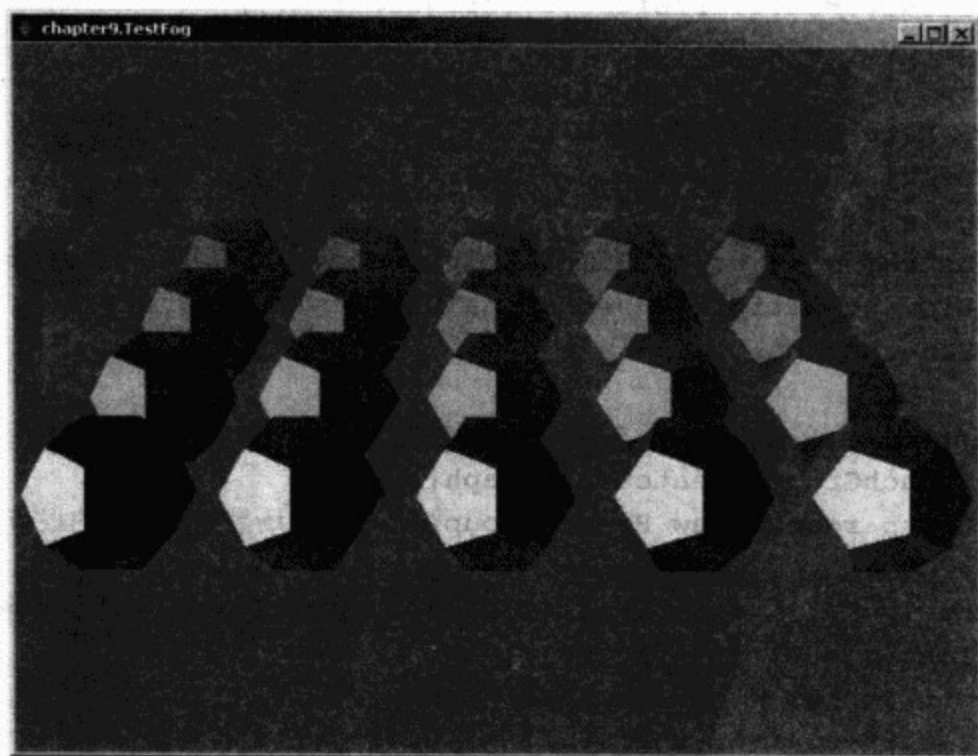


图9-12 带有大气衰减的雾场景

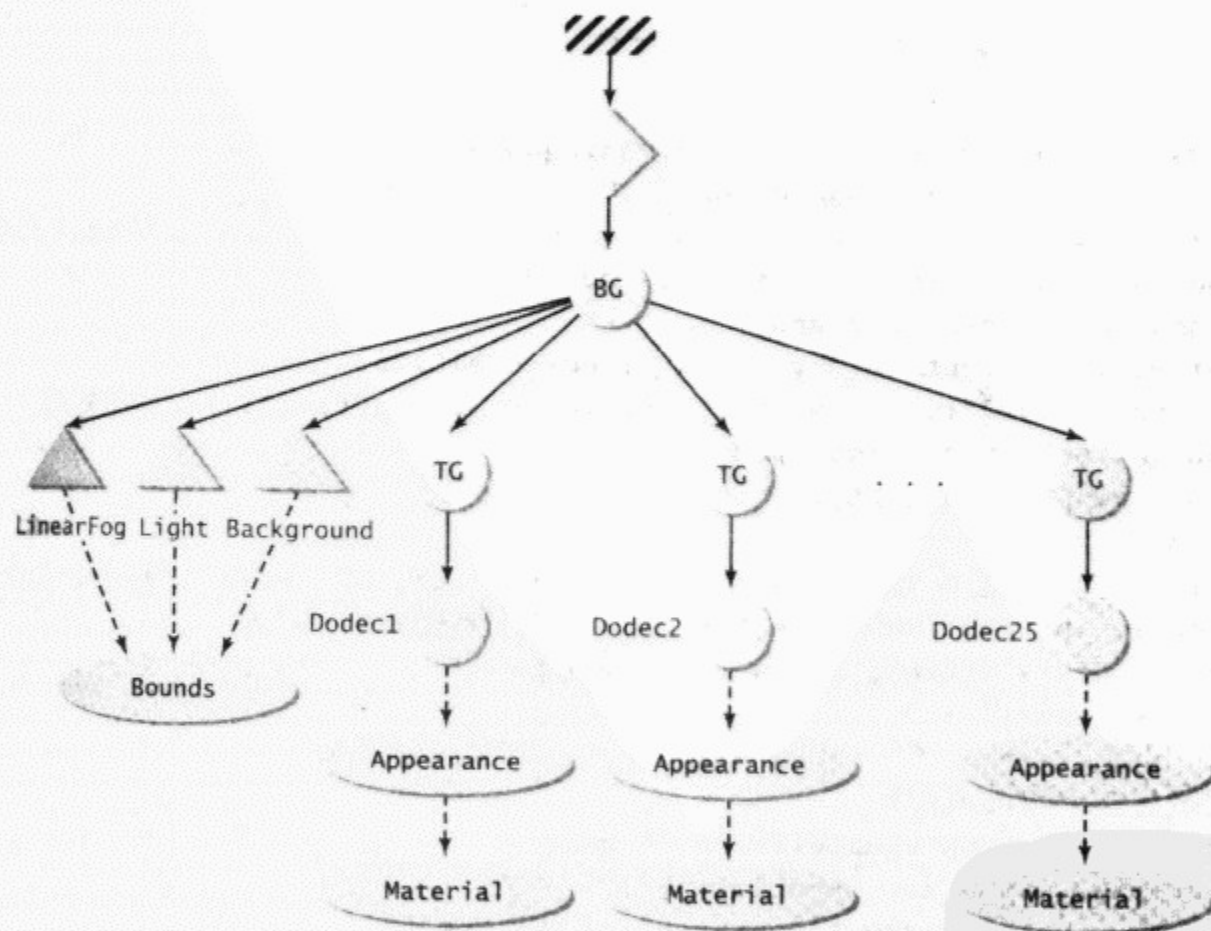


图9-13 雾实例的场景图

一组25个十二面体以 5×5 的网格形式放置在场景中（第35~42行）。createShape方法创建场景图的一个分支，该场景图中包含了TransformGroup类组节点、Dodecahedron1类对象以及Material的Appearance类对象。变换节点用来缩放十二面体，并且将其放到指定位置上。

用一个白色平行光源沿方向 $(1, 0, -1)$ 来照射这个场景，背景为灰色，将与背景颜色相同的线性雾节点加到场景图中（第54行）。线性雾的近距离是2，远距离为4，远处的对象消失在背景中。类Bounds的对象由light、background和fog三个类对象进行共享。

9.6 纹理映射

通常，实际生活中的对象包含许多细微的地方，用一般的几何结构为所有的细节建立模型，会很快耗尽计算机的资源。另一方面，数字图像在描述一些复杂的细节时，相对来说开销并不大。纹理映射是一种利用图像进行图形绘制的方法，它能够有效地提供大量的模型细节。

9.6.1 创建2D纹理映射

2D纹理映射是一种将2D图像映射到3D对象表面的方法。在纹理映射的过程中，我们将纹理图像上的映射点称为纹元 (texel)，而将3D对象表面上的映射点称为像素 (pixel)。纹理图像有它自己的坐标，纹理映射通常将表面每个顶点与纹元的纹理坐标一一对应。通过插值方法，可以将表面上的其他像素映射到纹理坐标上。显然，在一般情况下，像素与纹元之间的映射不是一一对应关系。如图9-14所示，一个像素可能只对应于一个纹元的一部分，也可能覆盖多个纹元，前者称为扩大 (magnification) 而后者称为缩小 (minification)。无论是哪种情况，我们都需要建立一些规则来获得纹理值，这些规则称为放大滤波器和缩小滤波器。常用的滤波器有最近纹元的选取和相邻纹元的线性插值 (组合)。

297

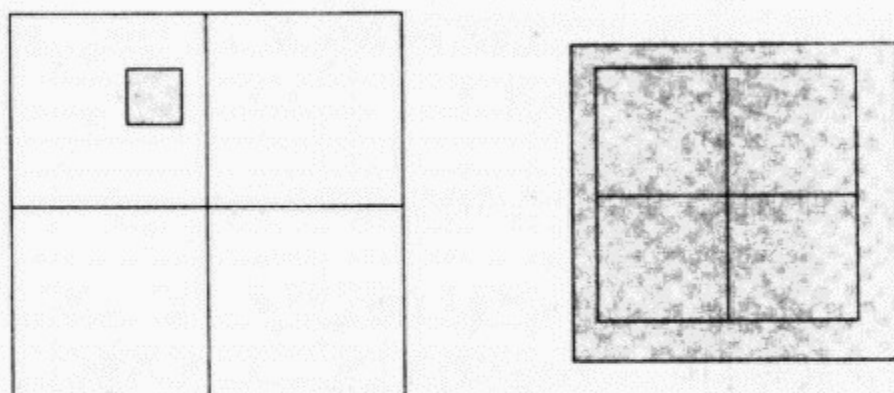


图9-14 扩大与缩小。灰色格子表示一个像素

Java 3D通过几何体的纹理坐标设置以及外观对象中的纹理对象，来支持纹理映射。在一个模型上生成纹理，需要经历两个步骤：

- 1) 将纹理坐标分配到几何体的顶点。
- 2) 在外观包 (appearance bundle) 上创建一个Texture类对象。

GeometryArray类及其子类便于指定每个顶点的纹理坐标，纹理坐标为顶点定义纹元的位置。Texture2D类定义用于纹理映射的纹理图像，它是一个节点组件，并且可以由Appearance类对象进行引用。

在构造一个GeometryArray类对象时，可以为几何体的顶点指定纹理坐标，这可以像坐标、颜色和法线等其他属性一样进行指定，2D纹理坐标的常量标记是TEXTURE_COORDINATE_2。

纹理一般是从图像中获取的。Texture2D类是表示纹理的NodeComponent类的子类。类ImageComponent2D表示用于Texture2D类的实际图像，Texture2D类对象引用ImageComponent2D类的对象。在Texture2D中，用于设置纹理图像的方法如下：

```
public void setImage(int index, ImageComponent2D) // 按照index指定的模式来设置2D纹理对象的纹理图像
```

ImageComponent2D对象可以由一个BufferedImage类或RenderedImage类的对象来获取它的图像，但RenderedImage类用得更为普遍：

```
public void set(BufferedImage bi)
public void set(RenderedImage ri)
```


用第4章中所描述的方法，可以从头创建或从文件载入BufferedImage类对象。例如，ImageIO类包含一些用于读取图像的静态函数。用下面的程序代码段来说明这一过程：

298

```
BufferedImage bi=ImageIO.read(file);
ImageComponent2D image=
new ImageComponent2D(ImageComponent2D.FORMAT_RGB, bi);
Texture2D texture = new Texture2D();
texture.setImage(0, image);
```

也可以用工具类TextureLoader从一个文件或BufferedImage对象中载入一幅纹理图像，所载入的图像可以以一个ImageComponent2D类对象的形式返回，或者直接就可以返回一个Texture类的对象。下述代码段显示用TextureLoader类的对象载入纹理图像的基本步骤：

```
TextureLoader loader = new TextureLoader(filename, this);
// 返回类Texture的对象
Texture texture = loader.getTexture();
// 或者首先返回图像
ImageComponent2D image = loader.getImage();
Texture2D texture2D = new Texture2D();
Texture2D.setImage(0, image);
```

利用setTexture方法，可以将已经创建的Texture2D对象添加到一个Appearance类对象中：

```
Appearance appear = new Appearance();
appear.setTexture(texture);
```

在几何体中设置好纹理坐标，并将纹理图像放置到Appearance类中后，就可以对该对象进行纹理映射。

程序清单9-4演示了2D纹理的映射过程。程序的运行结果显示一个经过纹理映射处理的地球仪，几何对象是一个旋转的球体。一幅地图用做纹理，并将它映射到这个球体的表面，该方法绘制了一个具有真实感的3D地球仪。

程序清单9-4 TextureMapping.java

```
1 package chapter9;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.net.*;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.image.*;
11 import java.applet.*;
12 import com.sun.j3d.utils.applet.MainFrame;
13 //定义TextureMapping类，继承自Applet类，演示2D纹理映射
14 public class TextureMapping extends Applet {
15     public static void main(String[] args) {
16         new MainFrame(new TextureMapping(), 640, 480); //创建主窗口并设置大小
17     }
18     //重写初始化方法
19     public void init() {
20         //创建画布
21         GraphicsConfiguration gc =
22             SimpleUniverse.getPreferredConfiguration();
```



```
23 Canvas3D cv = new Canvas3D(gc); //用gc构造Canvas3D对象
24 setLayout(new BorderLayout());
25 add(cv, BorderLayout.CENTER);
26 BranchGroup bg = createSceneGraph(); //创建场景分支子图
27 bg.compile();
28 SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
29 su.getViewingPlatform().setNominalViewingTransform();
30 su.addBranchGraph(bg);
31 }
32 //生成BranchGroup的私有方法, 创建场景分支图
33 private BranchGroup createSceneGraph() {
34     BranchGroup root = new BranchGroup(); //创建场景分支子图的根节点
35     TransformGroup spin = new TransformGroup();
36     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
37     root.addChild(spin);
38     //创建场景物体
39     Appearance ap = createTextureAppearance(); //生成具有纹理的外观对象
40     Sphere shape =
41     new Sphere(0.7f, Primitive.GENERATE_TEXTURE_COORDS, 50, ap);
42     spin.addChild(shape);
43     //旋转
44     Alpha alpha = new Alpha(-1, 6000);
45     RotationInterpolator rotator =
46     new RotationInterpolator(alpha, spin);
47     BoundingSphere bounds = new BoundingSphere();
48     rotator.setSchedulingBounds(bounds);
49     spin.addChild(rotator);
50     //添加背景
51     Background background = new Background(1.0f, 1.0f, 1.0f);
52     background.setApplicationBounds(bounds);
53     root.addChild(background);
54     return root;
55 }
56 //创建纹理表面
57 Appearance createTextureAppearance(){
58     Appearance ap = new Appearance();
59     URL filename = //图像文件URL
60     getClass().getClassLoader().getResource("images/earth.jpg");
61     TextureLoader loader = new TextureLoader(filename, this); //纹理载入
62     ImageComponent2D image = loader.getImage();
63     if(image == null) {
64         System.out.println("can't find texture file.");
65     }
66     Texture2D texture = new Texture2D
67     (Texture.BASE_LEVEL, Texture.RGBA,
68     image.getWidth(), image.getHeight());
69     texture.setImage(0, image); //设置纹理的图像
70     texture.setEnabled(true);
71     texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
72     texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
73     ap.setTexture(texture); //设置纹理
74     return ap;
75 }
76 }
```

299

该程序运行结果的屏幕截图如图9-15所示，图9-16给出了该程序的场景图。

创建的球体具有纹理坐标。在球体类Sphere的构造函数中，格式标志Primitive.GENERATE_TEXTURE_COORDS表示允许为构造的图元生成纹理坐标(第41行)。我们用RotationInterpolator类对象使球体发生旋转。

纹理外观是用createTextureAppearance方法来创建的(第57行)。通过TextureLoader类可以从磁盘文件中读出地球表面图像，然后将该图像分配给一个ImageComponent2D类的对象，用该对象可以创建一个Texture2D的对象。纹理通过与其相关联的Appearance类对象，就可以映射到球体的表面。放大滤波器和缩小滤波器都设置为线性插值。

300

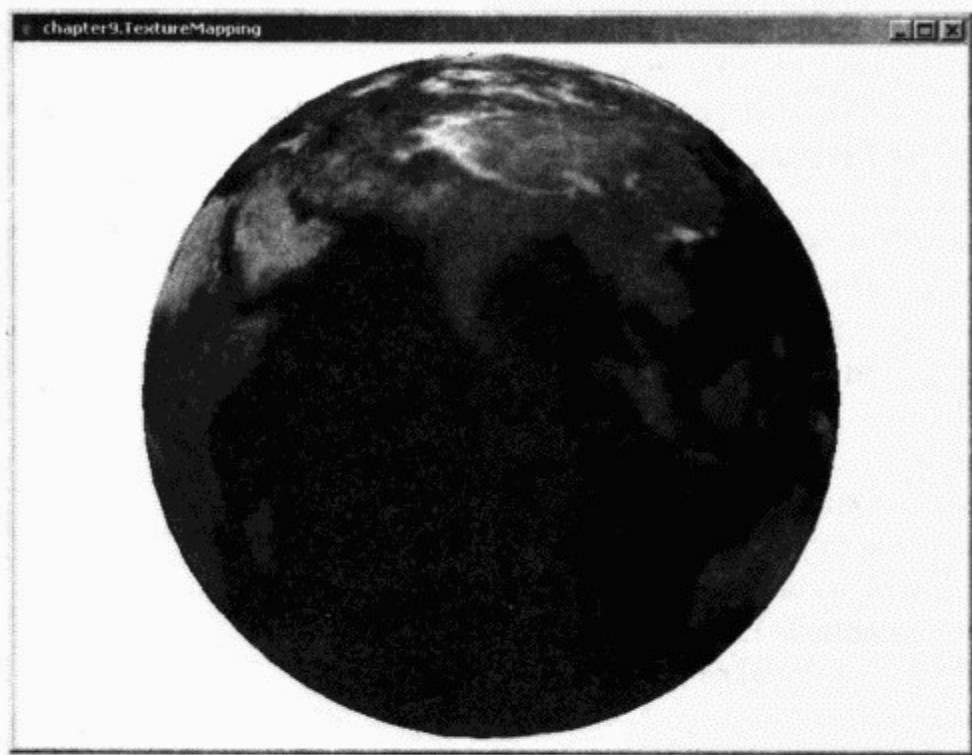


图9-15 带有纹理映射的旋转地球仪

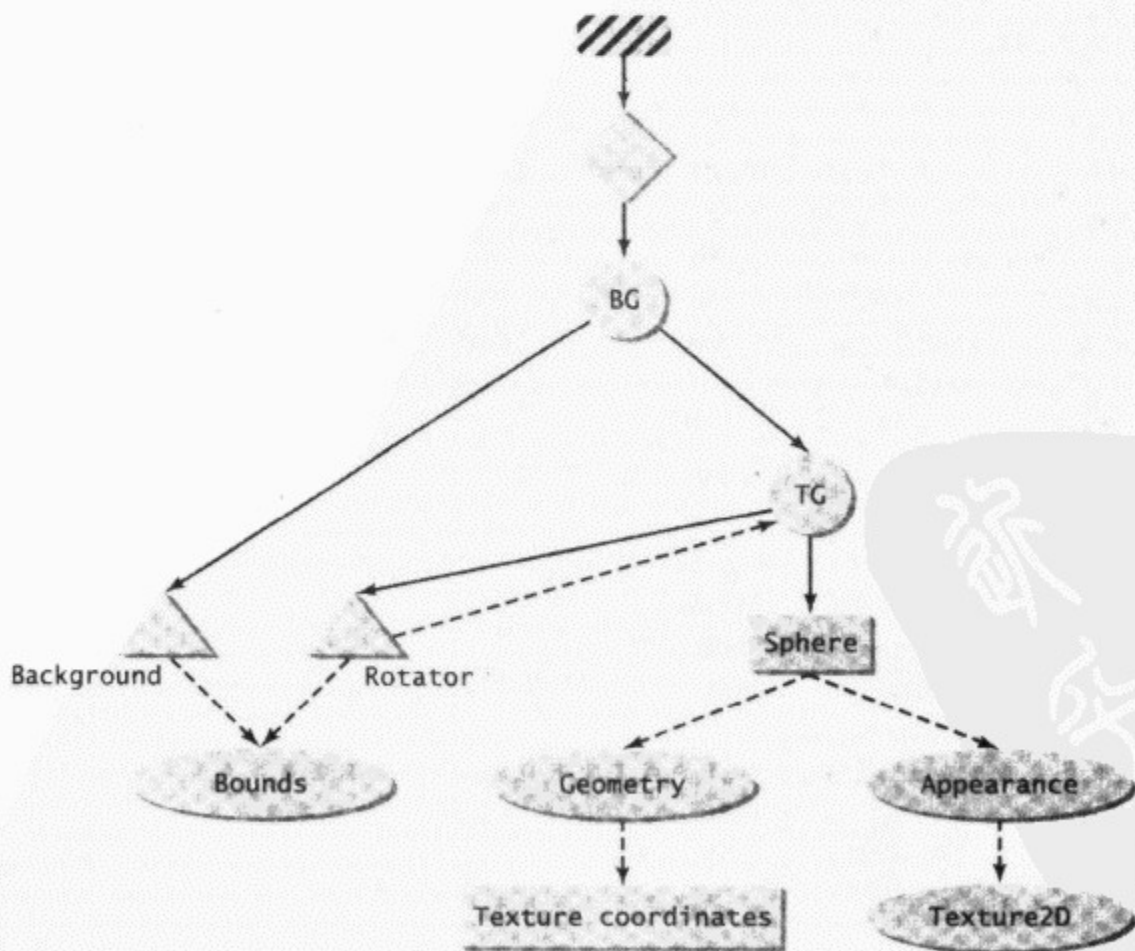


图9-16 纹理映射实例的场景图

301

9.6.2 纹理坐标

我们用几何体上的纹理坐标来控制纹理图像到表面的映射。在GeometryArray类中，用来设置纹理坐标的方法有：

```
// 设置此GeometryArray对象中指定索引顶点所关联的纹理坐标（单个）
void setTextureCoordinate(int texSet, int index, float[] texCoord)
void setTextureCoordinate(int texSet, int index, TexCoord2f texCoord)
// 设置此GeometryArray对象中指定索引开始的顶点所关联的纹理坐标（多个）
void setTextureCoordinates(int texSet, int index, float[] texCoords)
void setTextureCoordinates(int texSet, int index, TexCoord2f[] texCoords)
```

在IndexedGeometryArray类及其子类中，可以用以下方法来设置纹理坐标的下标：

```
// 设置对象中指定纹理坐标集合的指定索引处（集合索引）顶点的纹理顶点索引（索引数组中的值）
void setTextureCoordinateIndex(int texSet, int index, int texCoordIdx)
// 设置对象中指定纹理坐标集合的指定索引开始的（集合索引）顶点的纹理顶点索引（索引数组中的值）
void setTextureCoordinateIndices(int texSet, int index, int[] texCoordIdx)
```

用来表示纹理坐标的TexCoord2f类是在javax.vecmath包中定义的，类似地，还有表示3D纹理坐标的TexCoord3f类，以及表示四维纹理坐标的TexCoord4f类。对一个物体，我们可以定义一组纹理坐标，在一幅纹理图像中，纹理坐标是一个指定位置的2D向量 (u, v) ， u 与 v 的取值范围是从0.0到1.0，图像的四个角标的纹理坐标如图9-17所示。

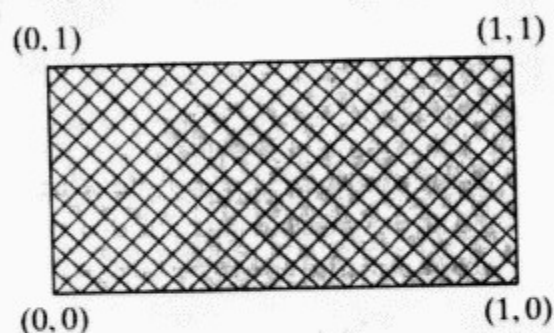


图9-17 用纹理坐标定义图像中的位置

我们来考虑一个问题，这就是要将如图9-18中指示的图像映射到一个立方体的表面，从而形成一个骰子。这幅图像可以分成6个正方形面，并且有12个截然不同的纹理坐标可以进行分配。为了实现所要求的纹理映射，应该将图像中正方形面的顶点纹理坐标分配给立方体各侧面的顶点。

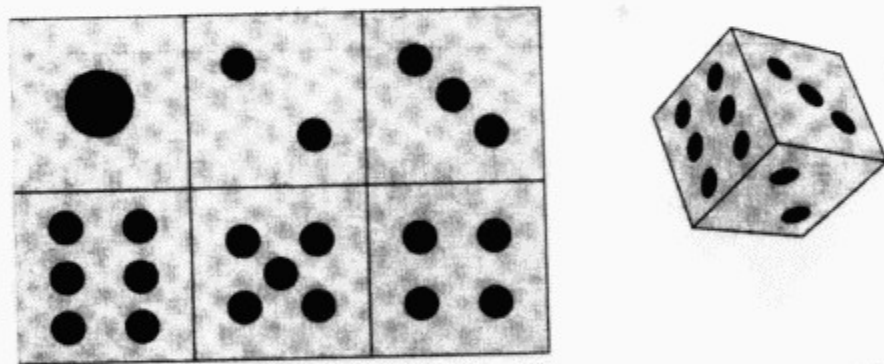


图9-18 将一幅图像映射到一个立方体的各个面

下面的代码段定义了这个带有合适的纹理坐标的立方体：

```
IndexedQuadArray qa = new IndexedQuadArray(12,
QuadArray.COORDINATES|QuadArray.TEXTURE_COORDINATE_2, 24);

Point3f[] coords = {new Point3f(0,0,0), new Point3f(1,0,0),
new Point3f(1,1,0), new Point3f(0,1,0),
new Point3f(0,0,1), new Point3f(1,0,1),
new Point3f(1,1,1), new Point3f(0,1,1)};
int[] coordIndices =
{0,1,2,3, 0,1,5,4, 1,2,6,5, 2,3,7,6, 3,0,4,7, 4,5,6,7};
qa.setCoordinates(0, coords);
qa.setCoordinateIndices(0, coordIndices);
```



```

TexCoord2f[] tex = {new TexCoord2f(0, 1), new TexCoord2f(1f/3, 1),
new TexCoord2f(2f/3, 1), new TexCoord2f(1, 1),
new TexCoord2f(0, 0.5f), new TexCoord2f(1f/3, 0.5f),
new TexCoord2f(2f/3, 0.5f), new TexCoord2f(1, 0.5f),
new TexCoord2f(0, 0), new TexCoord2f(1f/3, 0),
new TexCoord2f(2f/3, 0), new TexCoord2f(1, 0)};
int[] texIndices =
{0,1,5,4, 1,2,6,5, 2,3,7,6, 5,6,10,9, 6,7,11,10, 4,5,9,8};
qa.setTextureCoordinates(0,0,tex);
qa.setTextureCoordinateIndices(0,0,texIndices);

```

9.6.3 结合纹理映射与光照

可以将纹理映射和诸如光照模型之类的其他着色模型结合起来使用。Java 3D中有一个TextureAttributes类，它可以控制纹理映射的方式。例如，如果想要结合纹理映射和光照来绘制一个对象，那么可以创建一个配置模式来同时选用这两种方式，并且用一个TextureAttributes的类对象来混合这两种方式中的颜色。该图形的几何特征既有法线，也有纹理坐标集，在场景图中放置一些光源，并且在外观包用到一个Material类组件。在TextureAttributes类中，定义了多种不同的方法，用于结合纹理映射与着色方式。下述是如何将纹理映射和光照相结合的程序代码段：

```

TextureAttributes texatt = new TextureAttributes(); // 创建一个纹理属性
texatt.setTextureMode(TextureAttributes.COMBINE); // 设置纹理属性
appearance.setTextureAttributes(texatt);

```

程序清单9-5用来示例纹理映射和光照相结合的效果，该程序的运行结果是显示一个带有纹理映射并且接收光照的圆柱体（如图9-19所示）。该程序演示了如何设置几何体的纹理坐标，怎样在程序代码中创建一个纹理图像，以及如何用TextureAttributes类结合纹理映射与光照这三个过程。

程序清单9-5 Cup.java

```

1 package chapter9;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.geom.*;
6 import java.awt.image.*;
7 import java.net.URL;
8 import java.util.*;
9 import java.awt.event.*;
10 import javax.media.j3d.*;
11 import com.sun.j3d.utils.universe.*;
12 import com.sun.j3d.utils.geometry.*;
13 import com.sun.j3d.utils.image.*;
14 import java.applet.*;
15 import com.sun.j3d.utils.applet.MainFrame;
16 import com.sun.j3d.utils.behaviors.mouse.*;
17 //定义Cup类，继承自Applet类，演示纹理映射和光照的合成
18 public class Cup extends Applet {
19     public static void main(String[] args) {
20         new MainFrame(new Cup(), 640, 480); //创建主窗口并设置大小
21     }

```



```
22 //重写初始化方法
23 public void init() {
24     //创建画布
25     GraphicsConfiguration gc =
26         SimpleUniverse.getPreferredConfiguration();
27     Canvas3D cv = new Canvas3D(gc); //用gc构造Canvas3D对象
28     setLayout(new BorderLayout());
29     add(cv, BorderLayout.CENTER);
30     BranchGroup bg = createSceneGraph(); //创建场景分支图
31     bg.compile();
32     SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
33     su.getViewingPlatform().setNominalViewingTransform();
34     su.addBranchGraph(bg);
35 }
36 //生成BranchGroup的私有方法, 创建场景分支图
37 private BranchGroup createSceneGraph() {
38     BranchGroup root = new BranchGroup();
39     TransformGroup spin = new TransformGroup();
40     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
41     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
42     root.addChild(spin);
43     //object transform
44     Transform3D tr = new Transform3D();
45     tr.setScale(0.3);
46     tr.setRotation(new AxisAngle4d(0, 0, 1, Math.PI/12));
47     TransformGroup tg = new TransformGroup(tr);
48     spin.addChild(tg);
49     //创建场景物体
50     Geometry geom = createGeometry(); //调用函数创建场景的几何形体
51     Appearance ap = createTextureAppearance(); /* 生成场景物体表面*/
52     PolygonAttributes pa =
53         new PolygonAttributes(PolygonAttributes.POLYGON_FILL,
54             PolygonAttributes.CULL_NONE, 0);
55     ap.setPolygonAttributes(pa);
56     Shape3D shape = new Shape3D(geom, ap);
57     tg.addChild(shape);
58     //旋转
59     BoundingSphere bounds = new BoundingSphere();
60     Alpha alpha = new Alpha(-1, 10000);
61     RotationInterpolator rotator = new RotationInterpolator
62         (alpha, spin);
63     rotator.setSchedulingBounds(bounds);
64     spin.addChild(rotator);
65     //添加背景和光源
66     Background background = new Background(1.0f, 1.0f, 1.0f);
67     background.setApplicationBounds(bounds);
68     root.addChild(background);
69     AmbientLight light = new AmbientLight(true,
70         new Color3f(Color.lightGray)); //添加浅灰色环境光源
71     light.setInfluencingBounds(bounds);
72     root.addChild(light);
73     PointLight ptlight = new PointLight(new Color3f(Color.white),
74         new Point3f(3f, 3f, 3f), new Point3f(1f, 0f, 0f)); //添加白色点光源
75     ptlight.setInfluencingBounds(bounds);
```

```

76     root.addChild(ptlight);
77     return root;
78 }
79 //创建场景几何形体
80 Geometry createGeometry() {
81     int m = 120;
82     int n = 2;
83     QuadArray qa = new QuadArray(4*m,
84     QuadArray.COORDINATES|QuadArray.NORMALS|
85     QuadArray.TEXTURE_COORDINATE_2);
86     //生成圆柱体
87     Transform3D trans = new Transform3D();
88     trans.rotY(2*Math.PI/m);
89     Point3f pt0 = new Point3f(1,1,0);
90     Point3f pt1 = new Point3f(1,-1,0);
91     Vector3f normal = new Vector3f(1,0,0);
92     for (int j = 0; j < m; j++) {
93         qa.setCoordinate(j*4, pt0);
94         qa.setCoordinate(j*4+1, pt1);
95         qa.setNormal(j*4, normal);
96         qa.setNormal(j*4+1, normal);
97         trans.transform(pt0);
98         trans.transform(pt1);
99         trans.transform(normal);
100        qa.setCoordinate(j*4+2, pt1);
101        qa.setCoordinate(j*4+3, pt0);
102        qa.setNormal(j*4+2, normal);
103        qa.setNormal(j*4+3, normal);
104        //设置纹理坐标
105        TexCoord2f tex0 = new TexCoord2f(j*1f/m, 1f);
106        TexCoord2f tex1 = new TexCoord2f(j*1f/m, 0f);
107        qa.setTextureCoordinate(0,j*4,tex0);
108        qa.setTextureCoordinate(0,j*4+1,tex1);
109        tex0 = new TexCoord2f((j+1)*1f/m, 1f);
110        tex1 = new TexCoord2f((j+1)*1f/m, 0f);
111        qa.setTextureCoordinate(0,j*4+2,tex1);
112        qa.setTextureCoordinate(0,j*4+3,tex0);
113    }
114    return qa;
115 }
116 //创建纹理外观
117 Appearance createTextureAppearance(){
118     Appearance ap = new Appearance();
119     BufferedImage bi = new BufferedImage(512,128,
120     BufferedImage.TYPE_INT_ARGB);
121     Graphics2D g2 = (Graphics2D)bi.getGraphics();
122     g2.setColor(Color.white); //设置当前颜色
123     g2.fillRect(0, 0, 512,128); //绘制填充矩形
124     g2.setFont(new Font("Serif", Font.BOLD, 48)); //设置图形字体
125     g2.setColor(new Color(200,0,0)); //设置字体颜色
126     g2.drawString("Java 3D",0,100); //绘制字符串
127     ImageComponent2D image =
128         new ImageComponent2D(ImageComponent2D.FORMAT_RGBA, bi);
129     Texture2D texture = new Texture2D

```



```
130     (Texture.BASE_LEVEL, Texture.RGBA,  
131     image.getWidth(), image.getHeight());  
132     texture.setImage(0, image);  
133     texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);  
134     ap.setTexture(texture); //设置表面纹理  
135     //将纹理与光照相结合  
136     TextureAttributes texatt = new TextureAttributes();  
137     texatt.setTextureMode(TextureAttributes.COMBINE);  
138     ap.setTextureAttributes(texatt); //设置表面的纹理属性  
139     ap.setMaterial(new Material()); //设置表面的材质  
140     return ap;  
141 }  
142 }
```

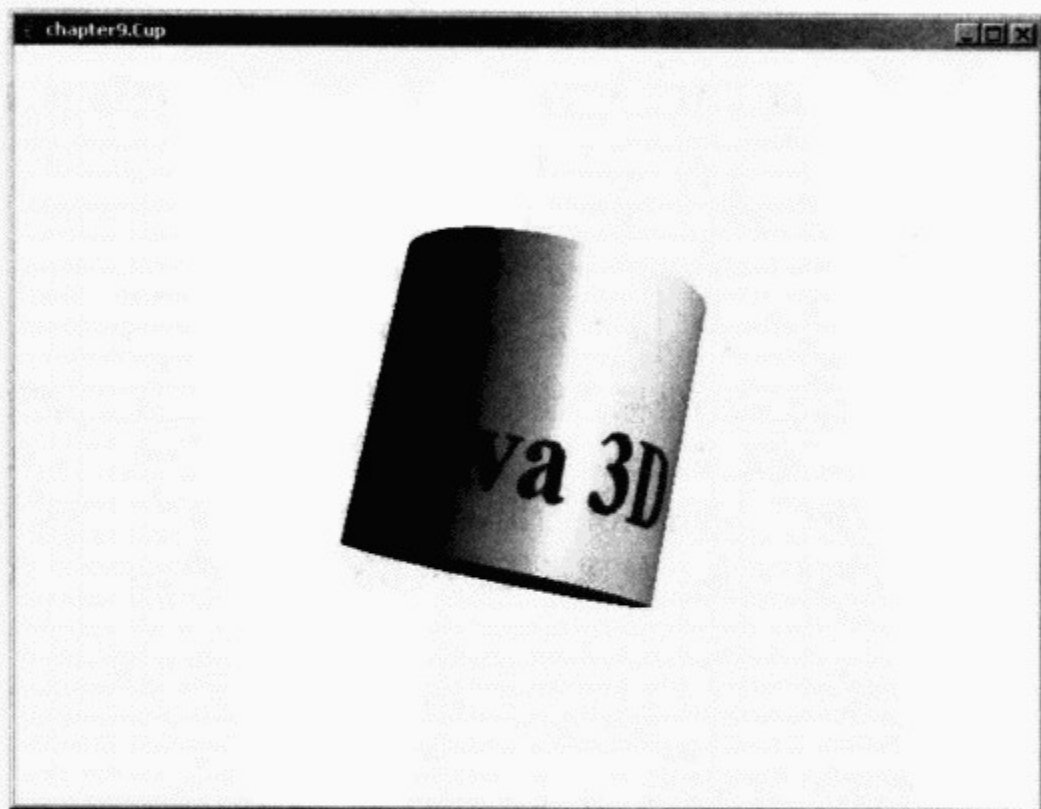


图9-19 一个带有纹理映射并接受光照的圆柱体

createGeometry方法用一个QuadArray类来构造圆柱体（第80行），法线和纹理坐标都是在这个几何体中生成的。

createTextureAppearance方法构造一个外观包的对象，它将一个Material类组件、一个Texture2D类组件和一个TextureAttributes类对象包括在内。用于纹理映射的图像直接创建一个BufferedImage类对象，其大小为512×128（第119行），返回Graphics2D类的一个对象，以允许在这幅图像上进行绘制。程序首先将该图像着色成白色，然后用红色字体绘制字符串“Java 3D”（第121~126行）。

用一个环境光源和一个点光源来照射这个圆柱体，将外观中的TextureAttributes类对象设置为COMBINE模式，从而可以将纹理映射与光照的效果结合起来（第137行）。

9.6.4 纹理坐标生成

在一个几何体中，可以直接指定纹理坐标，也可以运用一个基于对象几何特征和视图配置的模型，来自动生成纹理坐标。通过在外观中的一个特殊节点组件，可以实现纹理坐标的自动生成。纹理坐标既可以手动指定，也可以自动生成，这与对象着色方式相似：在对象几何特征中，可以直接指定颜色或者用光照模型自动生成颜色。

除了提供2D纹理坐标外, Java 3D还支持3D和四维纹理坐标, 这些纹理坐标的形式分别是 (r, s) 、 (r, s, t) 、 (r, s, t, q) 。这三种类型的纹理坐标(Texturecoordinate)的常量标志分别定义为:

```
TEXTURE_COORDINATE_2
TEXTURE_COORDINATE_3
TEXTURE_COORDINATE_4
```

这三种类型的纹理坐标都可以自动生成, 节点组件类TexCoordGeneration有助于自动生成纹理坐标。将TexCoordGeneration类的对象添加到外观包的对象中, 就可以自动生成纹理坐标。TexCoordGeneration类基于不同的标准, 定义了五种模式的纹理坐标生成方式:

```
OBJECT_LINEAR
EYE_LINEAR
SPHERE_MAP
NORMAL_MAP
REFLECTION_MAP
```

OBJECT_LINEAR和EYE_LINEAR这两种模式, 是用下述矩阵方程计算纹理坐标的:

$$\begin{bmatrix} r \\ s \\ t \\ q \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

因此, 对象的纹理坐标与对象的坐标线性相关。在OBJECT_LINEAR模式下, 点坐标 (x, y, z, w) 由对象坐标给出, 因此纹理是附属于该对象的, 并且随着对象移动而移动。在EYE_LINEAR模式下, 点坐标由视点坐标系给出, 因此纹理相对于观察者是固定的。当对象运动时, 纹理并不随着它动, 这样一来, 纹理可能看起来像是对象的影子。上述矩阵中的系数可以用下述方法进行设置:

```
void setPlaneR(Vector4f coeff)
void setPlaneS(Vector4f coeff)
void setPlaneT(Vector4f coeff)
void setPlaneQ(Vector4f coeff)
```

每一种方法设置矩阵中的一行系数。

SPHERE_MAP模式生成的纹理坐标, 可用于模拟基于视点坐标系的球面反射, 最后两种模式仅仅可以用于特殊纹理的立方体映射。一般的Texture2D类只定义一种矩形平面纹理。TextureCubeMap类定义了含有六幅图像的纹理, 这六幅图像将映射到立方体的六个面。纹理映射将每幅图像映射到对象的某个特定侧面, 对一些3D立体来说, 这种映射可能会更方便。在TextureCubeMap类对象中, 可以用如下常量来识别这六幅图像:

```
NEGATIVE_X
NEGATIVE_Y
NEGATIVE_Z
POSITIVE_X
POSITIVE_Y
POSITIVE_Z
```

307

程序清单9-6用TextureCubeMap类演示纹理坐标生成的不同模式。程序显示了一个旋转的十二面体, 其纹理由一个TextureCubeMap类对象定义。在纹理映射中, 三幅图像用于立方体的六个面, 相对的两个面所采用的图像相同(如图9-20所示)。用一个TexCoordGeneration类对象来生成十二面体的纹理坐标, 五个按钮以纹理坐标的生成模式作为标签。当点击一个按钮时, 就会为TexCoordGeneration类选择相应的模式来绘制十二面体。

程序清单9-6 CubeTexture.java

```
1 package chapter9;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.net.*;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.image.*;
11 import chapter6.Dodecahedron;
12 import java.applet.*;
13 import com.sun.j3d.utils.applet.MainFrame;
14 //定义CubeTexture类,继承自Applet类,演示使用TextureCubeMap生成纹理坐标
15 public class CubeTexture extends Applet implements ActionListener{
16     public static void main(String[] args) {
17         new MainFrame(new CubeTexture(), 640, 480); //创建主窗口并设置大小
18     }
19
20     private Appearance ap = null;
21     //重写初始化方法
22     public void init() {
23         setLayout(new BorderLayout()); //设置布局管理器
24         Panel panel = new Panel(); //创建放置按钮的面板
25         panel.setLayout(new GridLayout(5,1));
26         add(panel, BorderLayout.EAST);
27         Button button = new Button("OBJECT_LINEAR"); //加入各按钮及事件侦听器
28         button.addActionListener(this);
29         panel.add(button);
30         button = new Button("EYE_LINEAR");
31         button.addActionListener(this);
32         panel.add(button);
33         button = new Button("SPHERE_MAP");
34         button.addActionListener(this);
35         panel.add(button);
36         button = new Button("NORMAL_MAP");
37         button.addActionListener(this);
38         panel.add(button);
39         button = new Button("REFLECTION_MAP");
40         button.addActionListener(this);
41         panel.add(button);
42
43         GraphicsConfiguration gc =
44             SimpleUniverse.getPreferredConfiguration();
45         Canvas3D cv = new Canvas3D(gc); //用gc构造Canvas3D对象
46         add(cv, BorderLayout.CENTER);
47         BranchGroup bg = createSceneGraph(); //创建场景分支子图
48         bg.compile();
49         SimpleUniverse su = new SimpleUniverse(cv); //创建设置SimpleUniverse对象
50         su.getViewingPlatform().setNominalViewingTransform();
51         su.addBranchGraph(bg);
52     }
```

```

53 //生成BranchGroup的私有方法、创建场景分支图
54 private BranchGroup createSceneGraph() {
55     BranchGroup root = new BranchGroup();
56     TransformGroup spin = new TransformGroup();
57     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
58     root.addChild(spin);
59     //创建场景物体
60     ap = createTextureAppearance();
61     Shape3D shape = new Dodecahedron();
62     shape.setAppearance(ap);
63     Transform3D tr = new Transform3D();
64     tr.setScale(0.4);
65     TransformGroup tg = new TransformGroup(tr);
66     tg.addChild(shape);
67     spin.addChild(tg);
68     //旋转
69     Alpha alpha = new Alpha(-1, 18000);
70     RotationInterpolator rotator = new RotationInterpolator
71         (alpha, spin);
72     BoundingSphere bounds = new BoundingSphere();
73     rotator.setSchedulingBounds(bounds);
74     spin.addChild(rotator);
75     //背景
76     Background background = new Background(1.0f, 1.0f, 1.0f);
77     background.setApplicationBounds(bounds);
78     root.addChild(background);
79     return root;
80 }
81 //创建纹理外观
82 Appearance createTextureAppearance(){
83     Appearance ap = new Appearance();
84     URL filename = //图片URL
85         getClass().getClassLoader().getResource("images/earth.jpg");
86     TextureLoader loader = new TextureLoader(filename, this); //载入图片
87     ImageComponent2D image1 = loader.getImage(); //分别载入三幅图片
88     filename = getClass().getClassLoader().getResource
89         ("images/stone.jpg");
90     loader = new TextureLoader(filename, this);
91     ImageComponent2D image2 = loader.getImage();
92     filename = getClass().getClassLoader().getResource
93         ("images/sky.jpg");
94     loader = new TextureLoader(filename, this);
95     ImageComponent2D image3 = loader.getImage();
96
97     TextureCubeMap texture =
98         new TextureCubeMap(Texture.BASE_LEVEL, Texture.RGBA,
99             image1.getWidth());
100     texture.setImage(0, TextureCubeMap.NEGATIVE_X, image3);
101     texture.setImage(0, TextureCubeMap.NEGATIVE_Y, image1);
102     texture.setImage(0, TextureCubeMap.NEGATIVE_Z, image2);
103     texture.setImage(0, TextureCubeMap.POSITIVE_X, image3);
104     texture.setImage(0, TextureCubeMap.POSITIVE_Y, image1);
105     texture.setImage(0, TextureCubeMap.POSITIVE_Z, image2);
106
107     texture.setEnable(true);

```



```

108 texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
109 texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
110 ap.setTexture(texture);
111 //纹理坐标生成
112 TexCoordGeneration tcg = new
113 TexCoordGeneration(TexCoordGeneration.OBJECT_LINEAR,
114 TexCoordGeneration.TEXTURE_COORDINATE_3);
115 tcg.setPlaneR(new Vector4f(2, 0, 0, 0));
116 tcg.setPlaneS(new Vector4f(0, 2, 0, 0));
117 tcg.setPlaneT(new Vector4f(0, 0, 2, 0));
118 ap.setTexCoordGeneration(tcg);
119 ap.setCapability(Appearance.ALLOW_TEXGEN_WRITE);
120 return ap;
121 }
122 //事件响应函数
123 public void actionPerformed(ActionEvent e) {
124     String cmd = e.getActionCommand();
125     TexCoordGeneration tcg = new TexCoordGeneration();
126     if ("OBJECT_LINEAR".equals(cmd)) {
127         tcg.setGenMode(TexCoordGeneration.OBJECT_LINEAR);
128     } else if ("EYE_LINEAR".equals(cmd)) {
129         tcg.setGenMode(TexCoordGeneration.EYE_LINEAR);
130     } else if ("SPHERE_MAP".equals(cmd)) {
131         tcg.setGenMode(TexCoordGeneration.SPHERE_MAP);
132     } else if ("NORMAL_MAP".equals(cmd)) {
133         tcg.setGenMode(TexCoordGeneration.NORMAL_MAP);
134     } else if ("REFLECTION_MAP".equals(cmd)) {
135         tcg.setGenMode(TexCoordGeneration.REFLECTION_MAP);
136     }
137     tcg.setFormat(TexCoordGeneration.TEXTURE_COORDINATE_3);
138     tcg.setPlaneR(new Vector4f(2, 0, 0, 0));
139     tcg.setPlaneS(new Vector4f(0, 2, 0, 0));
140     tcg.setPlaneT(new Vector4f(0, 0, 2, 0));
141     ap.setTexCoordGeneration(tcg);
142 }
143 }

```

309

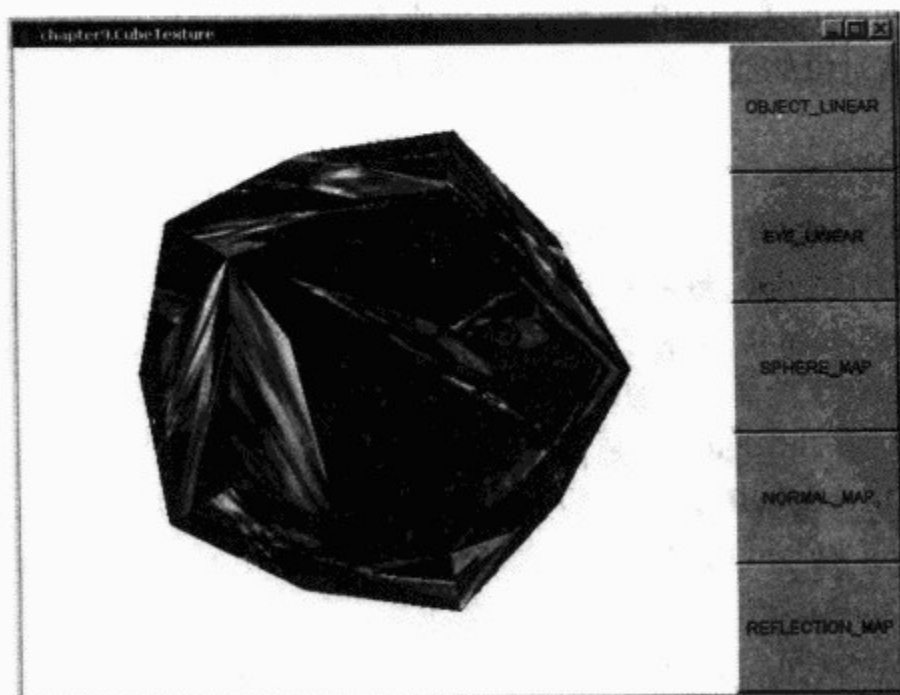


图9-20 用不同纹理坐标生成模式来生成立方体纹理

310

场景图如图9-21所示。一个十二面体附属两个TransformGroup节点：其中一个用于旋转，另一个用于对对象进行缩放操作。此十二面体对象有一个Appearance类对象，引用一个TextureCubeMap类对象和一个TexCoordGeneration类对象。设置外观的能力比特ALLOW_TEXGEN_WRITE，可以在实时场景图中改变TexCoordGeneration（第119行）。

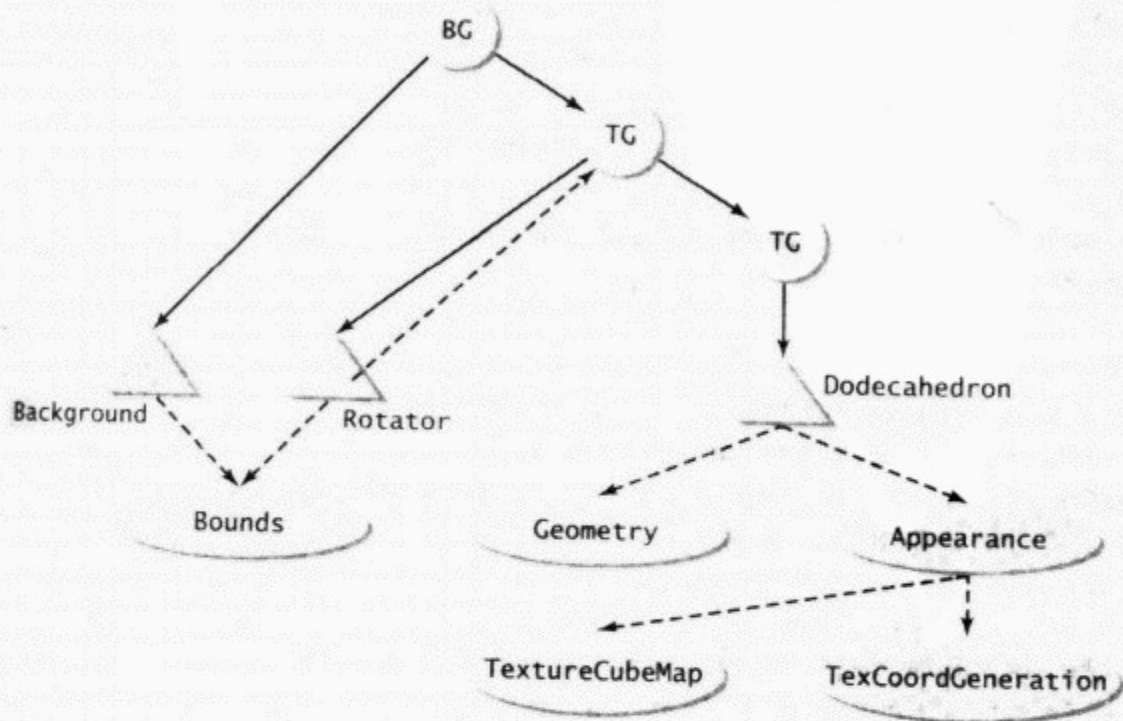


图9-21 纹理坐标实例的场景图

从图像文件中读出三幅大小相同的图像（第84~95行），TextureCubeMap类的对象的六个面将会用到这些图像，相对的两个面使用的图像一样。

使用TexCoordGeneration类的对象，可以自动生成TextureCubeMap类所需要的纹理坐标，默认的坐标生成方式设置为OBJECT_LINEAR。

该程序的主窗口使用一个布局管理器类BorderLayout，将Canvas3D类的对象添加到“CENTER”区域。创建五个按钮，分别表示五种纹理坐标生成方式：OBJECT_LINEAR、EYE_LINEAR、SPHERE_MAP、NORMAL_MAP及REFLECTION_MAP。将这些按钮添加到一个位于applet窗口“东”区的面板中，这些按钮将该applet窗口注册为监听器。在事件处理函数actionPerformed（第123行）中，我们新建一个TexCoordGeneration类对象。在返回按钮的标签后，就可以用函数setGenMode设置相应的纹理坐标生成方式。通过调用函数setTexCoordGeneration后，就可以将Appearance类对象的纹理坐标生成方式变为新建的TexCoordGeneration类对象。

主要的类和方法

- javax.vecmath.TexCoord2f 封装了2D纹理坐标的类。
- javax.vecmath.TexCoord3f 封装了3D纹理坐标的类。
- javax.vecmath.TexCoord4f 封装了四维纹理坐标的类。
- javax.media.j3d.Light 封装了光源的类。
- javax.media.j3d.AmbientLight 封装了环境光源的类。
- javax.media.j3d.DirectionalLight 封装了平行光源的类。
- javax.media.j3d.PointLight 封装了点光源的类。
- javax.media.j3d.SpotLight 封装了聚光光源的类。
- javax.media.j3d.Material 为光照定义材质属性的类。
- javax.media.j3d.Fog 封装了绘制效果的类。

- `javax.media.j3d.LinearFog` 混合系数为线性方程的雾类。
- `javax.media.j3d.ExponentialFog` 混合系数为指数方程的雾类。
- `javax.media.j3d.Texture2D` 封装了2D纹理的类。
- `javax.media.j3d.ImageComponent2D` 封装了图像的类。
- `javax.media.j3d.TextureLoader` 用于载入图像的类。
- `javax.media.j3d.TextureAttributes` 用于控制纹理映射属性的外观组件。
- `javax.media.j3d.TextureCubeMap` 在立方体表面用六幅图像定义其纹理的类。
- `javax.media.j3d.TexCoordGeneration` 封装了纹理坐标生成方式的类。

关键术语

- **Phong 模型 (Phong Model)** 一种考虑了表面法线、光线及人眼方向的光照模型。
- **环境光 (ambient light)** 在所有方向上都有均一的背景光照的模型。
- **方向光 (directional light)** 一种沿固定方向发出平行光线的光源。
- **点光 (point light)** 一种从一个固定点发出光线的光源。
- **聚光光源 (spotlight)** 一种光线发射角受到限制的点光源。
- **材质 (material)** 影响颜色计算的对象反射属性。
- **环境反射 (ambient reflection)** 对环境光源的反射。
- **漫反射 (diffuse reflection)** 对方向光源和点光源在各方向都一样地进行反射。
- **镜面反射 (specular reflection)** 对方向光源和点光源集中在一个固定方向上的反射。
- **景深效果处理 (depth cueing)** 将远处的物体变得模糊,造成该物体融入到背景中的视觉效果。
- **纹理映射 (texture mapping)** 一种将一幅图像映射到一个3D几何体表面的绘制技术。
- **纹元 (texel)** 纹理的一个基本单元。
- **纹理坐标 (texture coordinate)** 一种在几何体中使用的坐标,用来指定纹元的映射位置。

本章提要

- 在本章中,可以了解一些在3D图形绘制中如何实现具有真实感的物体外观的方法。
- 光照模型基于物体的物理特征和光源来计算物体的颜色。Phong模型是一种常用光照模型。Java 3D通过光源对象、几何学方面的表面法向量和外观属性中的材质,提供对这种光照模型的支持。
- 计算机图形学中,有四种常见的光源:环境光源、方向光源、点光源和聚光光源。
- 对于环境发射、漫反射及镜面反射来说,材质属性是用反射系数来描述的,对每一个基色都要指定这些系数。与光波长无关的亮度指数,也用来描述镜面反射的集中程度。
- 大气衰减和景深效果处理是一种通过调和远处物体来提高真实感的方法。Java 3D提供Fog类及其子类LinearFog和ExponentialFog,来实现这个效果。
- 纹理映射是另外一种可以很好地增强所绘制物体真实感的方法,纹理映射使用图像来绘制图形表面的细节。为了在Java 3D中应用纹理映射,必须在可视对象的几何体中定义纹理坐标,并且在其表面包含一幅纹理图像。纹理坐标可以直接用几何形式进行指定,或者可以用一个TexCoordGeneration对象来自动生成外观的纹理坐标。

312

复习题

- 9.1 一个点光源的衰减系数是 (1,2,1)。当离光源距离为多少时,衰减为0.04?
- 9.2 试讨论Phong光照模型公式在点光源和方向光源的条件下有什么不同?
- 9.3 一个点光源在表面上一点发生镜面反射,请问人眼在什么地方观察到的反射是最强的?
- 9.4 一个对象上一坐标为 (1,0,0) 的点的法线方向为 (0,1,0),一个光线强度为1.0的方向光源放在坐标为 (1,-1,0) 的位置,眼睛的位置是 (5,3,0)。如果该对象的RGB漫反射系数是 (0.3,0.5,0.2),请计算该

点漫反射的RGB值。

- 9.5 在习题9.4所给的场景中，如果材料的RGB镜面反射系数是(1,1,0.5)并且亮度是10，请计算该点镜面反射的RGB值。
- 9.6 一个对象上一点(1,0,0)的表面法线方向为(0,1,0)，一个光线强度为1.0的点光源放在坐标为(0,1,0)的位置，眼睛的位置是(5,3,0)。如果该对象的RGB漫反射系数是(0.3,0.5,0.2)，请计算该点漫反射的RGB值。
- 9.7 在习题9.6所给的场景中，如果材料的RGB镜面反射系数是(1,1,0.5)并且亮度是10，请计算该点镜面反射的RGB值。
- 9.8 在线性雾方程中，当距离是多少时，混合系数 f 为0.5?
- 9.9 在指数雾方程中，当距离是多少时，混合系数 f 为0.5?
- 9.10 请问纹理图像中点的纹理坐标是多少?
- 9.11 在一个TexCoordGeneration类对象中，系数矩阵设为多少时，可以将原本的纹理图像本身映射到物体表面?

编程练习

- 9.1 编写一个程序，显示Java 3D的四种着色模式。在场景中创建四个正方形，每个正方形使用其中一种着色方式。
- 9.2 修改程序清单9-1，用于显示一个3D文本字符串。
- 9.3 编写一个与程序清单9-2相似的程序，但要求镜面反射系数不同。
- 9.4 使用一个线性雾对象在场景中显示一个很长的长方形，要能观察到该长方形的远处边消失在背景中。
- 9.5 使用一个指数雾对象修改程序清单9-3，添加一个选项以允许用户在程序运行时，可以修改雾的密度参数。
- 9.6 编写一个Java 3D程序，用于显示一个经过纹理映射处理的四面体，用同样的纹理图像映射到四面体的各个面。将四面体各个面上三个顶点的纹理坐标，设置为与纹理图像的三个角点相对应，要求该四面体在场景中进行旋转运动。
- 9.7 用OBJECT_LINEAR纹理坐标生成方式，重写练习9.6中的程序。
- 9.8 用EYE_LINEAR纹理坐标生成方式，重写练习9.7中的程序。
- 9.9 重写程序清单9-4，用于显示一个受到红色聚光光源照射并带有纹理映射的地球仪。
- 9.10 编写一个Java 3D程序，用于显示一个正在旋转的骰子，用一个TextureCubeMap类对象构造一个立方体。在程序中，通过在BufferedImage类对象上画圆圈生成六幅图像，用一个TexCoordGeneration类对象自动设置纹理坐标，使用OBJECT_LINEAR作为纹理坐标的生成方式。

第10章 行为和交互

学习目标

- 理解图形中的动态行为。
- 理解Java 3D的Behavior类和WakeupCondition类。
- 在场景图中应用Behavior节点。
- 使用鼠标行为。
- 使用键盘导向行为。
- 使用视图平台行为。
- 对拾取和行为进行合并。

315

10.1 引言

现代计算机图形系统已不仅限于绘制静态场景，在3D场景中引入动态变化，是许多图形应用中的重要部分。计算机图形中有两种常见的动态变化：交互和动画。交互（interaction）是根据用户的输入改变图形场景。动画（animation）生成随着时间变化的图形绘制序列，产生动态效果。这两种机制在图形系统中都产生了动态的行为。变化可以发生在图形对象的虚拟世界或“看到的”这个世界的视图中。变化可能涉及可视系统的几何属性、外观属性、变换以及其他方面。一般来说，动态行为的合并可能是一个复杂的过程，通常需要使用一些程序代码来处理这些动态变化。比较好的做法是，在图形程序设计范型中系统地引入动态逻辑。

Java 3D用“行为”这个概念来实现场景图中的动画、交互以及其他动态机制。一个Behavior（行为）对象是场景图中的叶节点，它定义了当行为被激活（唤醒）时所要执行的动作。行为是由称之为唤醒条件的特殊对象进行触发的，如果唤醒条件与移动鼠标之类的用户动作相关，则该行为就定义了一种交互形式。如果行为是由与时间有关的唤醒条件触发的，则该行为就定义的是一种动画形式。

在这一章中，我们将介绍行为的一般概念，以及一种常见的行为类型：交互。我们还将讨论拾取功能在交互定义方面的应用情形。动画是另一种重要的行为类型，相关内容将在第11章中加以讨论。

10.2 行为

Java 3D提供了一种统一的方法来实现动画和交互，它利用了Java的面向对象程序设计的优点，同时通过使用Behavior类层次结构以及其他相关类，向场景图中引入了动画和交互逻辑。Behavior类的类层次结构如图10-1所示。

Behavior类是一个抽象类，它继承了Leaf类。Behavior类定义了一般行为的基本框架，Behavior类有两个抽象方法：

```
void initialize( )  
void processStimulus(Enumeration wakeupCriteria)
```

构造Behavior对象时，要调用Behavior类的initialize方法。在特定的唤醒条件下，Java 3D会

316 调用Behavior类的processStimulus方法，Behavior类的子类重写了这两个方法，以完成特定的行为任务。

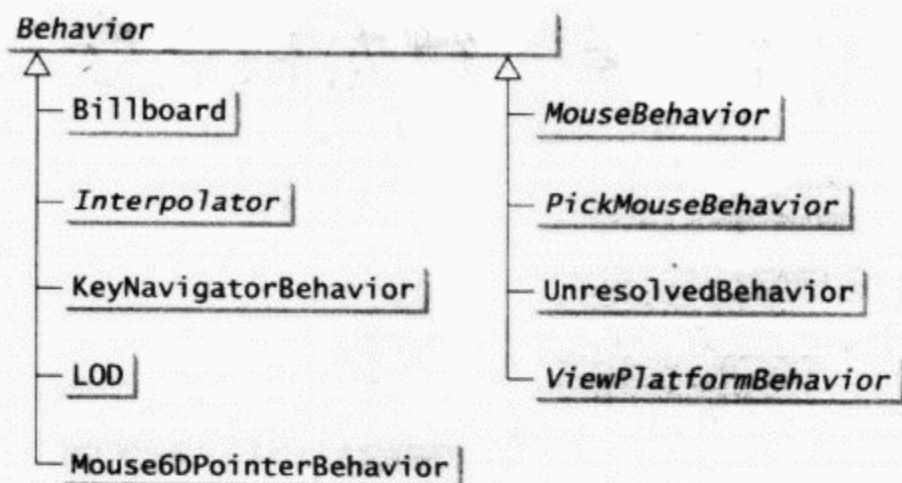


图10-1 Behavior类

Behavior对象和WakeupCondition对象合作完成其执行周期。Behavior类包含了下面的方法，用于设置触发行为的唤醒条件：

```
void wakeupOn(WakeupCondition wakeup)
```

图10-2演示了Behavior对象的典型执行周期。

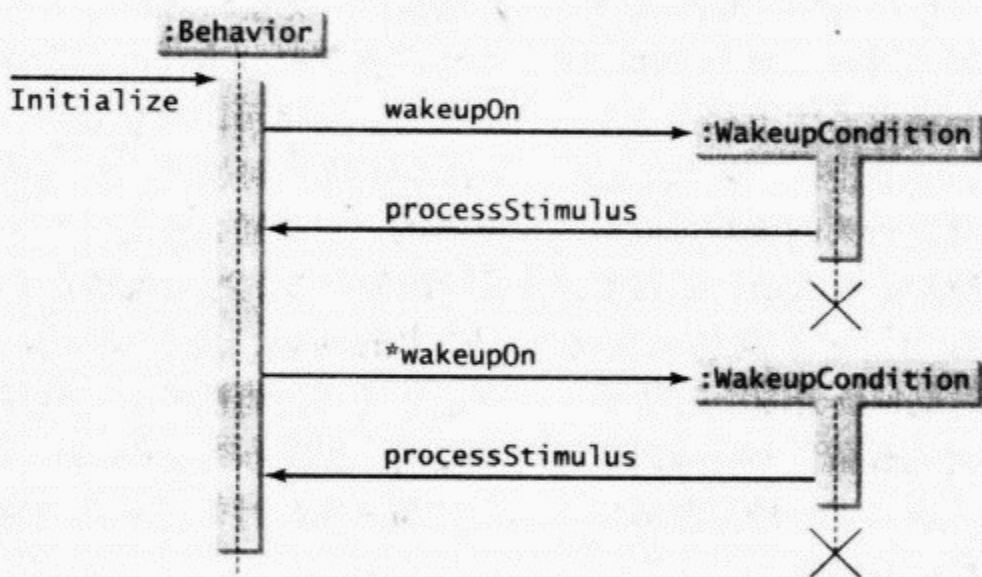


图10-2 行为和唤醒条件对象之间的交互

在Behavior对象的初始化过程中，将设置WakeupCondition对象。当指定的唤醒条件出现时，WakeupCondition对象将通过调用其processStimulus方法唤醒Behavior对象。processStimulus方法的参数WakeupCriteria是触发行为的一个wakeupCriteria对象列表。在执行了processStimulus方法中的自定义代码之后，通过调用wakeupOn方法，可再次设置唤醒条件，这一过程将继续无限地循环下去。

317 唤醒条件（Wakeup conditions）是动态行为的重要部分，它们的作用是作为不同条件下触发不同行为的信号。Java 3D针对各种激励行为，给出了一整套唤醒标准。同时，还提供了通过逻辑运算符以不同方式组合使用这些标准的能力。WakeupCondition类的类层次结构如图10-3所示。

WakeupCriterion的子类代表了能触发Behavior对象的特定事件和标准。例如，WakeupOnElapsedTime定义了在经过一段固定的时间之后，触发行为的唤醒条件，WakeupOnAWTEvent启用像移动和点击鼠标等与AWT事件相关的唤醒条件。常见唤醒标准类的构造函数如下所示：

WakeupOnElapsedTime(long ms)

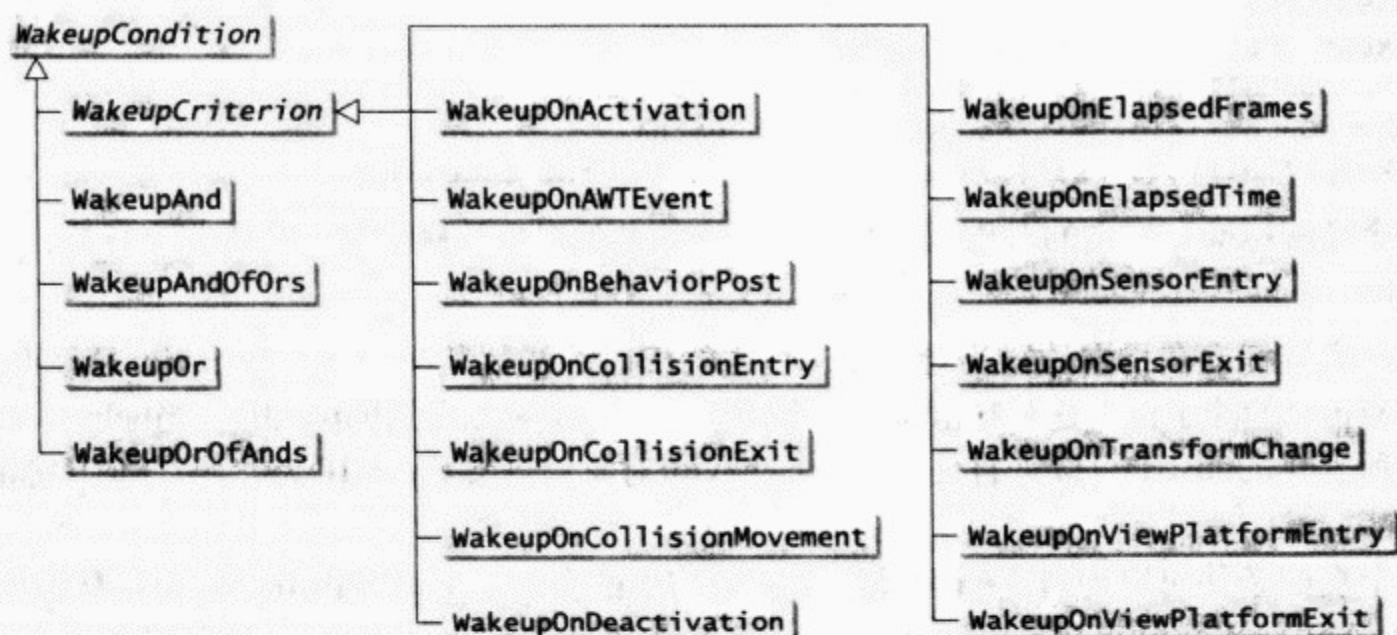


图10-3 触发条件类的层次关系

这个构造函数将构造一个唤醒标准，它在经历以毫秒为单位的给定时间间隔之后触发行为：

WakeupOnElapsedFrames(int frames)

这个构造函数将构造一个唤醒标准，它在指定的帧数之后触发行为：

WakeupOnAWTEvent(int eventID)

WakeupOnAWTEvent(long eventMask)

在指定的AWT事件发生时，这个唤醒标准会触发行为，AWT事件ID定义了一个AWT事件。例如下面的语句为鼠标键按下事件创建了一个标准：

```
new WakeupOnAWTEvent(Event.MOUSE_DOWN);
```

如果需要数个AWT事件，可以使用第二个带事件掩码参数的构造函数。AWT事件掩码可以用逻辑或（OR）操作进行合并。例如下面的唤醒标准与鼠标事件和鼠标移动事件相关联：

```
new WakeupOnAWTEvent(AWTEvent.MOUSE_EVENT_MASK |
    AWTEvent.MOUSE_MOTION_EVENT_MASK);
```

需要说明的是，Event类已经被AWTEvent类所代替：

WakeupOnTransformChange(TransformGroup node)

这个构造函数定义一个唤醒标准，用于当给定的TransformGroup结点中的变换发生改变时，唤醒某个行为：

```
WakeupOnCollisionEntry(Bounds bounds)
WakeupOnCollisionEntry(Node node)
WakeupOnCollisionEntry(Node node, int speedHint)
WakeupOnCollisionEntry(SceneGraphPath path)
WakeupOnCollisionEntry(SceneGraphPath path, int speedHint)
```

这个唤醒标准在指定对象与场景图中的其他对象发生碰撞时，唤醒某个行为：

```
WakeupOnCollisionMovement(Bounds bounds)
WakeupOnCollisionMovement(Node node)
WakeupOnCollisionMovement(Node node, int speedHint)
WakeupOnCollisionMovement(SceneGraphPath path)
WakeupOnCollisionMovement(SceneGraphPath path, int speedHint)
```

这些构造函数用于指定唤醒标准，即在指定的对象和场景图中的其他对象发生碰撞并移动

时，唤醒某个行为：

```
WakeupOnCollisionExit(Bounds bounds)
WakeupOnCollisionExit(Node node)
WakeupOnCollisionExit(Node node , int speedHint)
WakeupOnCollisionExit(SceneGraphPath path)
WakeupOnCollisionExit(SceneGraphPath path , int speedHint)
```

这个唤醒标准在指定的对象退出与场景图中的其他对象的碰撞时，唤醒某个行为：

```
WakeupOnBehaviorPost (Behavior behavior , int postID)
```

这种唤醒标准在指定的行为对象发送指定的ID时，激活某个行为。Behavior对象可以发送一个带有postID的事件，这种唤醒标准监视着由behavior对象发送的postID。当behavior参数为空时，所发送的postID可以来自任何一个Behavior对象。当参数postID为0时，任何ID都能满足这个唤醒标准。

多个唤醒条件可以通过逻辑运算组合成新的唤醒条件。WakeupCondition类有四个子类：WakeupAnd、WakeupOr、WakeupOrOfAnds和WakeupAndOfOrs，它们定义了布尔运算 (Boolean operators)，用于组合唤醒条件。每个类都有一个构造函数，这个构造函数以一组合适的对象为参数：

```
WakeupAnd(WakeupCriterion[] criteria)
WakeupOr(WakeupCriterion[] criteria)
WakeupOrOfAnds(WakeupAnd[] criteriaAnds)
WakeupAndOfOrs(WakeupOr[] criteriaOrs)
```

例如，下面的代码段定义了一个唤醒条件，指定的条件值为当时间过了10秒，或者绘制了5帧，或者发生碰撞：

```
WakeupCriterion[] criterial = {new WakeupOnElapsedTime(10000)};
WakeupCriterion[] criteria2 = {new WakeupOnElapsedFrames(5),
    new WakeupOnCollisionEntry(node)};
WakeupAnd[] ands = (new WakeupAnd(criterial),
    new WakeupAnd(criteria2));
WakeupOrOfAnds condition = new WakeupOrOfAnds(ands);
```

Behavior对象有一个作用范围边界。只有当其作用范围边界和视图平台 (view platform) 相交时，Behavior对象才起作用。可以使用下面的方法设置作用范围边界：

```
SetSchedulingBounds(Bounds bounds);
SetSchedulingBoundingLeaf(BoundingLeaf bounds);
```

实现一个自定义行为 (custom behavior) 的过程总结如下：

- 定义Behavior类的一个子类，重写其initialize方法和processStimulus方法。
- 在initialize方法中设置适当的唤醒条件。如果需要，在processStimulus中可以再次设置唤醒条件。
- 创建一个自定义的Behavior类对象的实例，并作为叶节点加到场景图中。
- 为行为对象设置作用范围边界。

程序清单10-1给出了用自定义的Behavior类实现的一个应用程序，这个程序显示了一个给出系统时间的指针式时钟，时钟的指针是不断地变化的。一个自定义的行为类 (见程序清单10-2) 实现了基于当前时间对时钟的更新。

程序清单10-1 Clock.java

```
1 package chapter10;
2
```



```
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.util.*;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import java.applet.*;
11 import com.sun.j3d.utils.applet.MainFrame;
12 //定义Clock类, 继承自Applet类, 运行结果为一只走动的时钟
13 public class Clock extends Applet {
14     public static void main(String[] args) {
15         new MainFrame(new Clock(), 640, 480); // 创建主窗口并设置大小
16     }
17     //重写Applet的初始化方法
18     public void init() {
19         //创建canvas
20         GraphicsConfiguration gc =
21             SimpleUniverse.getPreferredConfiguration();
22         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D对象
23         setLayout(new BorderLayout()); //设置布局
24         add(cv, BorderLayout.CENTER);
25         BranchGroup bg = createSceneGraph(); //调用createSceneGraph方法创建场景图
26         bg.compile();
27         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
28         su.getViewingPlatform().setNominalViewingTransform();
29         su.addBranchGraph(bg);
30     }
31     //生成BranchGroup的私有方法, 用于创建场景图
32     private BranchGroup createSceneGraph() {
33         BranchGroup root = new BranchGroup();
34         //创建时钟的表面
35         Appearance apFace = new Appearance(); //物体外观对象
36         Material matFace = new Material(); //材质对象
37         matFace.setAmbientColor(new Color3f(0f, 0f, 0f)); // 设置周围颜色
38         matFace.setDiffuseColor(new Color3f(0.15f, 0.15f, 0.25f)); // 设置光照散射颜色
39         apFace.setMaterial(matFace);
40         Cylinder face = new Cylinder(0.6f, 0.01f,
41             Cylinder.GENERATE_NORMALS, 50, 2, apFace);
42         Transform3D tr = new Transform3D(); //设置变换节点
43         tr.rotX(Math.PI/2);
44         tr.setTranslation(new Vector3d(0, 0, -0.01));
45         TransformGroup tg = new TransformGroup(tr);
46         tg.addChild(face);
47         root.addChild(tg);
48         //创建时针
49         Appearance ap = new Appearance();
50         ap.setMaterial(new Material());
51         Shape3D shapeHour =
52             new Shape3D(createGeometry(0.4, 0.02, 0.02), ap); //生成时针形体
53         TransformGroup spinHour = new TransformGroup();
54         spinHour.addChild(shapeHour);
55         spinHour.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

```

56     root.addChild(spinHour); //加入时针节点
57     //创建分针
58     Shape3D shapeMin =
59         new Shape3D(createGeometry(0.5, 0.02, 0.02), ap); //生成分针形体
60     TransformGroup spinMin = new TransformGroup();
61     spinMin.addChild(shapeMin);
62     spinMin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
63     root.addChild(spinMin); //加入分针节点
64     //创建秒针
65     Shape3D shapeSec =
66         new Shape3D(createGeometry(0.5, 0.01, 0.01), ap); //生成秒针形体
67     TransformGroup spinSec = new TransformGroup();
68     spinSec.addChild(shapeSec);
69     spinSec.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
70     root.addChild(spinSec); //加入秒针节点
71     //加入行为节点
72     ClockBehavior rotator =
73         new ClockBehavior(spinHour, spinMin, spinSec); //创建ClockBehavior对象
74     BoundingSphere bounds = new BoundingSphere();
75     rotator.setSchedulingBounds(bounds);
76     root.addChild(rotator);
77     //设置光照
78     AmbientLight light =
79         new AmbientLight(true, new Color3f(Color.blue)); //添加蓝色环境光源
80     light.setInfluencingBounds(bounds);
81     root.addChild(light);
82     PointLight ptlight = new PointLight(new Color3f(Color.white),
83         new Point3f(0.7f, 0.7f, 2f), new Point3f(1f, 0f, 0f)); //添加白色点光源
84     ptlight.setInfluencingBounds(bounds);
85     root.addChild(ptlight);
86     //设置背景
87     Background background = new Background(0.7f, 0.7f, 0.7f); //设置背景颜色
88     background.setApplicationBounds(bounds);
89     root.addChild(background); //加入背景节点
90     return root;
91 }
92 //生成GeometryArray对象, 生成时针、分针和秒针的方法
93 GeometryArray createGeometry(double l, double w, double h) {
94     GeometryInfo gi =
95         new GeometryInfo(GeometryInfo.TRIANGLE_ARRAY);
96     Point3d[] pts = new Point3d[4]; //定义点坐标
97     pts[0] = new Point3d(0, 0, h);
98     pts[1] = new Point3d(-w, 0, 0);
99     pts[2] = new Point3d(w, 0, 0);
100    pts[3] = new Point3d(0, l, 0);
101    gi.setCoordinates(pts); //设定点坐标
102    int[] indices = {0, 1, 2, 0, 3, 1, 0, 2, 3, 2, 1, 3}; //定义点索引数组
103    gi.setCoordinateIndices(indices); //设定顶点索引
104    NormalGenerator ng = new NormalGenerator();
105    ng.generateNormals(gi); //生成法向量
106    return gi.getGeometryArray(); //返回形状
107 }
108 }

```


程序清单10-2 ClockBehavior.java

```
1 package chapter10;
2
3 import java.util.*;
4 import javax.media.j3d.*;
5 //定义ClockBehavior类, 继承自Behavior类, 演示时钟运行的效果
6 public class ClockBehavior extends Behavior {
7     TransformGroup tgH;//声明时针节点变量
8     TransformGroup tgM;//声明分针节点变量
9     TransformGroup tgS;//声明秒针节点变量
10
11     public ClockBehavior(TransformGroup transH,
12     TransformGroup transM, TransformGroup transS) {
13         tgH = transH;
14         tgM = transM;
15         tgS = transS;
16     }
17     //初始化函数
18     public void initialize() {
19         wakeupOn(new WakeupOnElapsedTime(500)); //500ms后唤醒
20     }
21
22     public void processStimulus(java.util.Enumeration enumeration) {
23         int hour = Calendar.getInstance().get(Calendar.HOUR);//获取系统当前小时值
24         int min = Calendar.getInstance().get(Calendar.MINUTE);//获取系统当前分钟值
25         int sec = Calendar.getInstance().get(Calendar.SECOND);//获取系统当前秒钟值
26         Transform3D tr = new Transform3D();//创建Transform3D对象
27         tr.rotZ(-Math.PI * (hour+min/60.0)/6.0);//更新时针
28         tgH.setTransform(tr);
29         tr.rotZ(-Math.PI * min /30.0);//更新分针
30         tgM.setTransform(tr);
31         tr.rotZ(-Math.PI * sec /30.0);//更新秒针
32         tgS.setTransform(tr);
33         wakeupOn(new WakeupOnElapsedTime(500)); //500ms后唤醒
34     }
35 }
```

321

程序生成的3D时钟如图10-4所示, 图10-5是这个场景图的树状结构图。

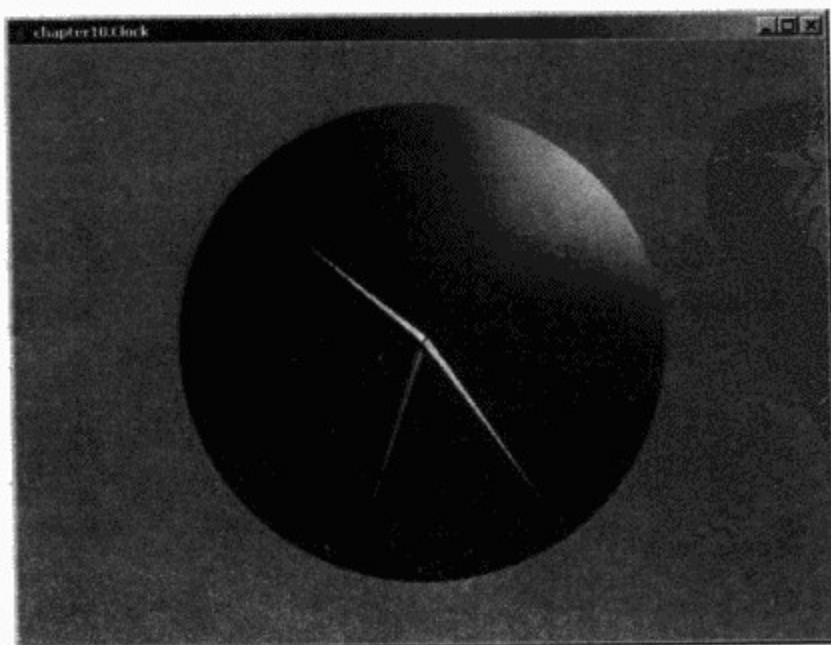


图10-4 Behavior对象驱动的时钟

这个程序显示了一个实时的指针式时钟，时针、分针、秒针都是由createGeometry方法（第93行）生成的。createGeometry方法用GeometryInfo生成了有四个顶点和四个三角面的几何体。表面法向量（Surface normals）由一个NormalGenerator对象生成，以辅助产生光照效果。在场景图中放置了一个蓝色环境光源和一个白色的点光源，背景颜色设置为灰色。

时钟的三个指针分别附属到一个TransformGroup节点，一系列变换控制着指针的转动。为了使时钟运转，我们还要周期性地更新这些变换，可以用一个行为对象来驱动更新活动。

322

ClockBehavior类是Behavior类的子类，它作用于控制着时钟指针的那些TransformGroup对象。在initialize方法（第19行）和processStimulus方法（第33行）中，设置每过0.5秒后就唤醒所设置的时钟行为，这个行为的主要任务是根据系统时间更新时钟指针的旋转。当前时间由Calender类得到，指针的旋转根据时间的计算得到：

$$\text{hourAngle} = -\frac{2\pi}{12} \left(\text{hour} + \frac{\text{minute}}{60} \right)$$

$$\text{minuteAngle} = -\frac{2\pi}{60} \text{minute}$$

$$\text{secondAngle} = -\frac{2\pi}{60} \text{second}$$

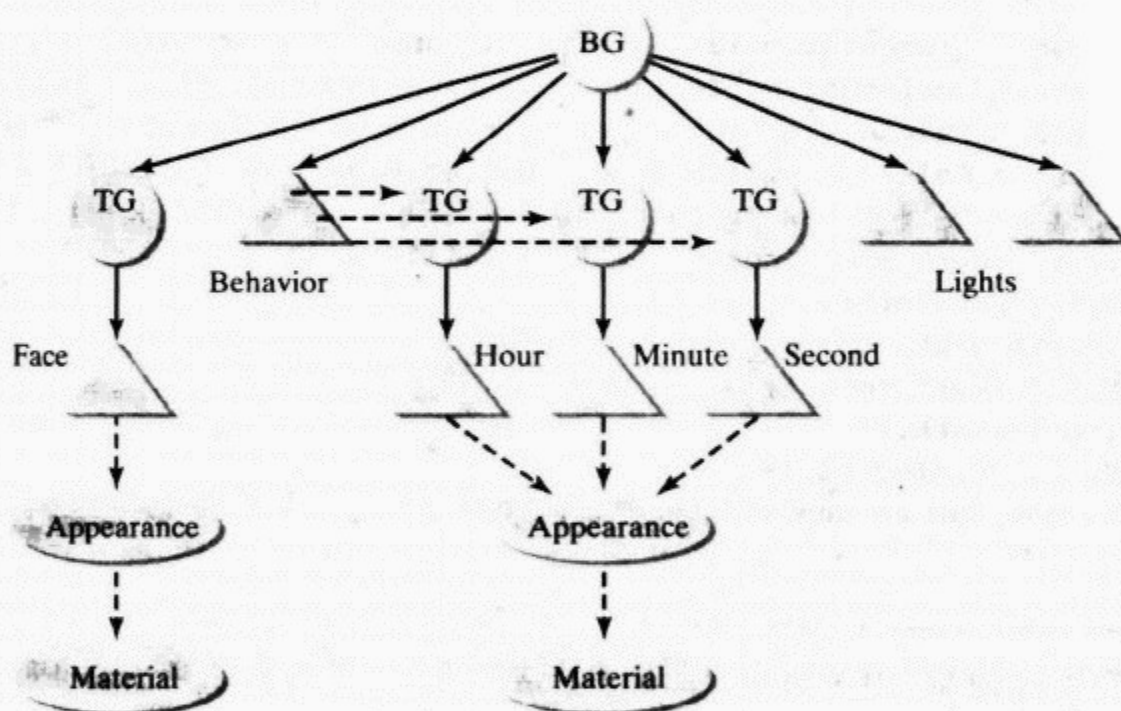


图10-5 时钟场景图

10.3 交互

交互处理的是由用户输入触发的动态行为。在行为框架下，交互和动画的一般机制是一样的，主要的差异是唤醒条件的来源不同。交互中的唤醒源是用户输入设备，比如鼠标、键盘以及3D输入设备等。

Java 3D为交互提供了多组行为子类。当然可以用基本的Behavior类和唤醒条件来构造自定义交互行为，预定义的工具类对常用的交互行为使用起来很方便。与交互相关的行为类如图10-6所示。

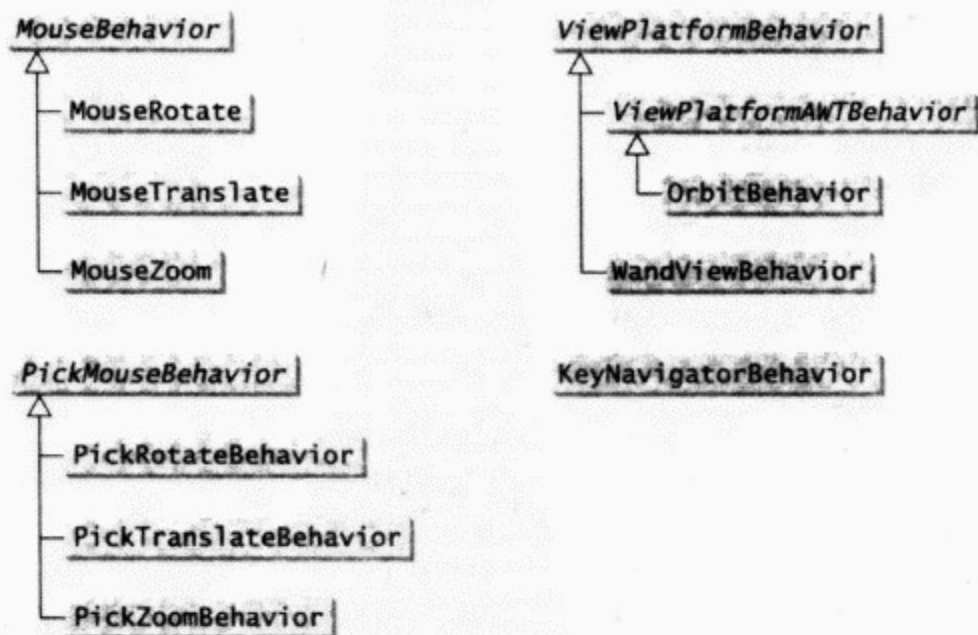


图10-6 与交互相关的类

10.3.1 鼠标行为

MouseBehavior类定义了与鼠标动作相关的特定行为，这一组行为都是在一个相关联的TransformGroup对象上进行操作，用户可以通过鼠标动作调整目标TransformGroup对象。这些行为中用到的鼠标动作包括以下几种类型：

- MouseRotate：鼠标左键按下时拖动鼠标。
- MouseTranslate：鼠标右键按下时拖动鼠标。
- MouseZoom：鼠标中键按下时拖动鼠标（或者在鼠标左键按下时按住Alt键）。

323

这三个类的用法是相似的，挑选了一个TransformGroup对象作为行为的目标对象。鼠标行为可以以两种方式和鼠标事件相关联：

- 1) 用WakeupOnAWTEvent来触发行为，通过Canvas3D上的鼠标事件来控制目标。
- 2) 指定一个AWT组件用于监听鼠标事件，用WakeupOnBehaviorPost来触发行为，这种模式在试图应用另一个AWT组件上的鼠标控制时很有用。

MouseBehavior类的构造函数包括：

```
MouseRotate(TransformGroup tg)
MouseRotate(Component c)
MouseRotate(Component c, TransformGroup tg)
MouseTranslate(TransformGroup tg)
MouseTranslate(Component c)
MouseTranslate(Component c, TransformGroup tg)
MouseZoom(TransformGroup tg)
MouseZoom(Component c)
MouseZoom(Component c, TransformGroup tg)
```

带Component参数的构造函数以第二种模式运行，它可以从另一个组件获得鼠标动作。

程序清单10-3是一个应用了各种MouseBehavior类的应用程序，它展示了一个地球仪和一组坐标轴（见图10-7），用户可以通过鼠标旋转、翻动、缩放地球仪，但坐标轴不受鼠标运动控制。

程序清单10-3 MoveGlobe.java

```
1 package chapter10;
2
3 import javax.vecmath.*;
```


324

```
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.net.URL;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.image.*;
11 import com.sun.j3d.utils.behaviors.mouse.*;
12 import chapter7.Axes;
13 import java.applet.*;
14 import com.sun.j3d.utils.applet.MainFrame;
15 //定义MoveGlobe类, 继承自Applet类, 演示了使用鼠标移动地球仪
16 public class MoveGlobe extends Applet {
17     public static void main(String[] args) {
18         new MainFrame(new MoveGlobe(), 480, 480); //创建主窗口并设定大小
19     }
20     //重写Applet的初始化函数
21     public void init() {
22         //创建canvas
23         GraphicsConfiguration gc =
24             SimpleUniverse.getPreferredConfiguration();
25         Canvas3D cv = new Canvas3D(gc);
26         setLayout(new BorderLayout());
27         add(cv, BorderLayout.CENTER);
28         TextArea ta = new TextArea("", 3, 30, TextArea.SCROLLBARS_NONE);
29         ta.setText("Rotation: Drag with left button\n"); //添加并设置文本域
30         ta.append("Translation: Drag with right button\n");
31         ta.append("Zoom: Hold Alt key and drag with left button");
32         ta.setEditable(false); //设置文本域不可编辑
33         add(ta, BorderLayout.SOUTH);
34         BranchGroup bg = createSceneGraph(); //调用createSceneGraph方法创建场景图
35         bg.compile();
36         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
37         su.getViewingPlatform().setNominalViewingTransform();
38         su.addBranchGraph(bg);
39     }
40     //生成BranchGroup的私有方法, 用于创建场景图
41     private BranchGroup createSceneGraph() {
42         BranchGroup root = new BranchGroup();
43         //创建坐标轴
44         Transform3D tr = new Transform3D();
45         tr.setScale(0.5);
46         tr.setTranslation(new Vector3d(-0.8, -0.7, -0.5));
47         TransformGroup tg = new TransformGroup(tr);
48         root.addChild(tg);
49         Axes axes = new Axes();
50         tg.addChild(axes); // 添加坐标轴节点
51         //设置变换
52         TransformGroup spin = new TransformGroup();
53         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
54         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
55         root.addChild(spin);
56         //设置纹理
57         Appearance ap = createAppearance();
```



```
58     spin.addChild(new Sphere(0.7f,
59     Primitive.GENERATE_TEXTURE_COORDS, 50, ap));
60     //设置鼠标转动
61     MouseRotate rotator = new MouseRotate(spin);
62     BoundingSphere bounds = new BoundingSphere();
63     rotator.setSchedulingBounds(bounds);
64     spin.addChild(rotator);
65     //设置鼠标平移
66     MouseTranslate translator = new MouseTranslate(spin);
67     translator.setSchedulingBounds(bounds);
68     spin.addChild(translator);
69     //设置鼠标缩放
70     MouseZoom zoom = new MouseZoom(spin);
71     zoom.setSchedulingBounds(bounds);
72     spin.addChild(zoom);
73     //设置光照
74     AmbientLight light =
75         new AmbientLight(true, new Color3f(Color.blue)); //添加蓝色环境光源
76     light.setInfluencingBounds(bounds);
77     root.addChild(light);
78     PointLight ptlight = new PointLight(new Color3f(Color.white),
79         new Point3f(0f, 0f, 2f), new Point3f(1f, 0.3f, 0f)); //添加白色点光源
80     ptlight.setInfluencingBounds(bounds);
81     root.addChild(ptlight);
82     //设置背景
83     Background background = new Background(1.0f, 1.0f, 1.0f); //设置背景颜色
84     background.setApplicationBounds(bounds);
85     root.addChild(background);
86     return root;
87 }
88 //创建外观方法，用平面图像建立3D纹理形状
89 Appearance createAppearance(){
90     Appearance appear = new Appearance(); //创建外观对象
91     URL filename =
92         getClass().getClassLoader().getResource("images/earth.jpg"); //获取平面图像
93     TextureLoader loader = new TextureLoader(filename, this); //纹理载入
94     ImageComponent2D image = loader.getImage();
95     Texture2D texture = new Texture2D
96         (Texture.BASE_LEVEL, Texture.RGBA,
97         image.getWidth(), image.getHeight());
98     texture.setImage(0, image);
99     texture.setEnabled(true);
100    texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
101    texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
102    appear.setTexture(texture);
103    return appear;
104 }
105 }
```

325

窗口中显示的是一个映射有纹理的地球仪和一组坐标轴，地球仪可以由鼠标动作控制，3D画布下的TextArea对象显示了鼠标操作的说明。当用户按着鼠标左键移动鼠标时，地球仪会转动，按住右键拖动鼠标使地球仪平移，鼠标左键按下时按住Alt键，拖动鼠标使地球仪缩放。

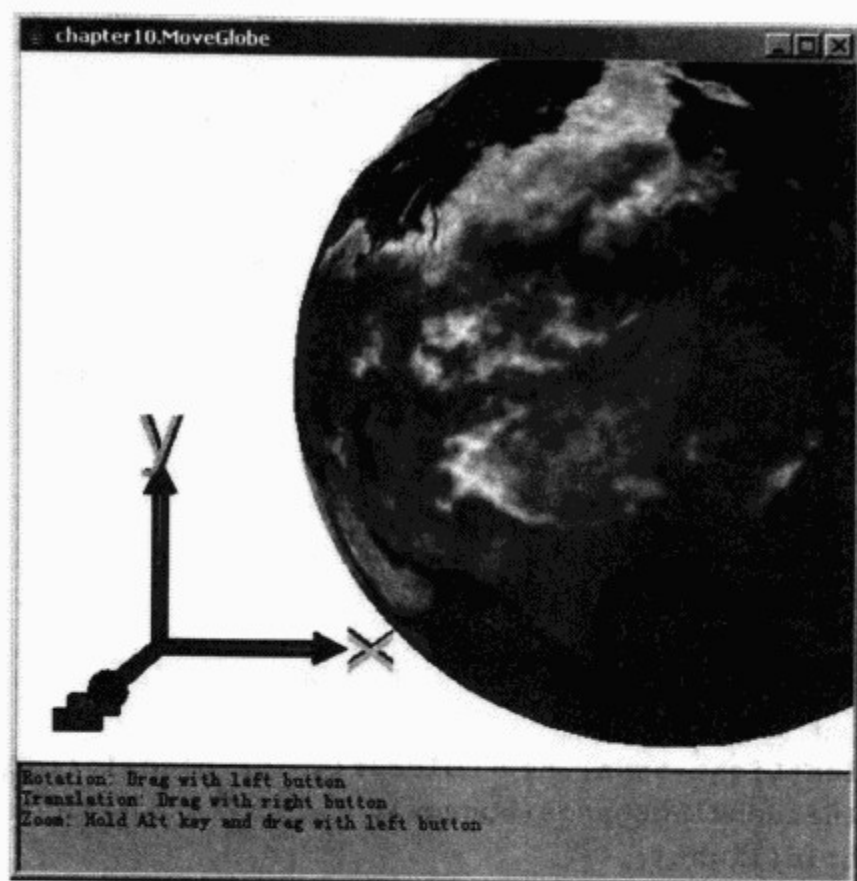


图10-7 用鼠标移动地球仪

程序的场景图如图10-8所示。Sphere对象附属到一个TransformGroup对象，场景图中放置了一个MouseRotate对象、一个MouseTranslate对象与一个MouseZoom对象，它们通过同一个TransformGroup对象作用在同一个球体上（第60~72行）。这三个行为对变换的不同部分进行调整，这三个行为共享了同一个边界对象作为作用范围边界。背景节点也引用这个边界对象。

Axes对象在场景图中的另外一个分支上，并且有它自己的TransformGroup对象，这个变换是不受鼠标行为控制的，因此，鼠标动作不会影响坐标轴。

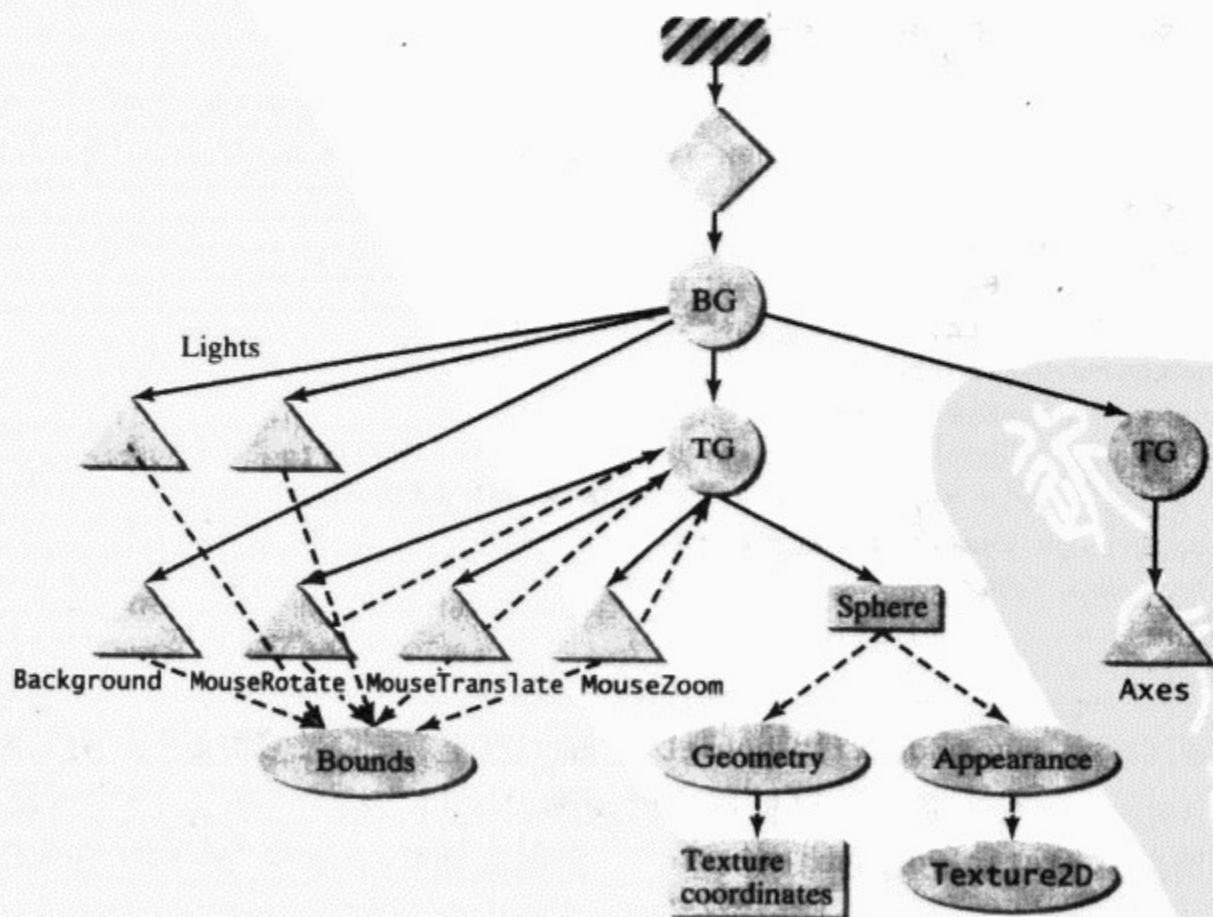


图10-8 鼠标行为示例的场景图

10.3.2 键盘行为

同鼠标行为相似，KeyNavigatorBehavior类在TransformGroup对象上进行操作，不过，行为是由键盘按键控制的。KeyNavigatorBehavior类有以下构造函数：

KeyNavigatorBehavior(TransformGroup tg)

KeyNavigatorBehavior(component c , TransformGroup tg)

KeyNavigatorBehavior类内建的按键控制（key controls）的定义如下：

- ←, →: 左/右旋转。
- ↑, ↓: 前/后平移。
- Alt-←, Alt-→: 左/右平移。
- PgUp, PgDn: 上/下转动。
- Alt-PgUp, Alt-PgDn: 上/下平移。
- - : 减少裁剪距离。
- + : 复位裁剪距离。
- = : 复位。

程序清单10-4给出了KeyNavigatorBehavior类的应用程序，它显示了一个带有纹理的地球仪，用户可以通过键盘来转动、平移和复位场景（见图10-9）。

程序清单10-4 TestKeyBehavior.java

```
1 package chapter10;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.net.URL;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.image.*;
11 import com.sun.j3d.utils.behaviors.keyboard.*;
12 import java.applet.*;
13 import com.sun.j3d.utils.applet.MainFrame;
14 //定义TestKeyBehavior类，继承自Applet类，演示键盘控制形体
15 public class TestKeyBehavior extends Applet {
16     public static void main(String[] args) {
17         new MainFrame(new TestKeyBehavior(), 480, 480); //创建主窗口并设置大小
18     }
19     //重写Applet初始化函数
20     public void init() {
21         GraphicsConfiguration gc =
22             SimpleUniverse.getPreferredConfiguration();
23         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D对象
24         setLayout(new BorderLayout()); //设置布局管理器
25         add(cv, BorderLayout.CENTER);
26         TextArea ta = new TextArea("", 3, 30, TextArea.SCROLLBARS_NONE);
27         ta.setText("Rotation: left, right, PgUp, PgDn\n"); //设置文本域
28         ta.append("Translation: up, down,
29             Alt-left, Alt-right, Alt-PgUp, Alt-PgDn\n");
30         ta.append("Reset: =\n");
```

326
327

```

31     ta.setEditable(false); //设置文本域不可编辑
32     add(ta, BorderLayout.SOUTH);
33     BranchGroup bg = createSceneGraph(); //创建场景图
34     bg.compile();
35     SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
36     su.getViewingPlatform().setNominalViewingTransform();
37     su.addBranchGraph(bg);
38 }
39 //生成BranchGroup的私有方法，用于创建场景图
40 private BranchGroup createSceneGraph() {
41     BranchGroup root = new BranchGroup();
42     TransformGroup spin = new TransformGroup(); //创建变换组节点并设置属性
43     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
44     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
45     root.addChild(spin);
46     //创建有纹理的地球仪
47     Appearance ap = createAppearance();
48     spin.addChild(new Sphere(0.7f,
49         Primitive.GENERATE_TEXTURE_COORDS, 50, ap));
50     //键盘行为
51     KeyNavigatorBehavior behavior = new KeyNavigatorBehavior(spin);
52     BoundingSphere bounds = new BoundingSphere();
53     behavior.setSchedulingBounds(bounds);
54     spin.addChild(behavior);
55     //设置背景
56     Background background = new Background(1.0f, 1.0f, 1.0f); //设置背景颜色
57     background.setApplicationBounds(bounds);
58     root.addChild(background);
59     return root;
60 }
61 //创建外观的方法，用平面图像建立3D纹理形状
62 Appearance createAppearance(){
63     Appearance appear = new Appearance(); //创建外观对象
64     URL filename =
65         getClass().getClassLoader().getResource("images/earth.jpg"); //获取平面图像
66     TextureLoader loader = new TextureLoader(filename, this); //纹理装载
67     ImageComponent2D image = loader.getImage();
68     Texture2D texture =
69         new Texture2D(Texture.BASE_LEVEL, Texture.RGBA,
70             image.getWidth(), image.getHeight());
71     texture.setImage(0, image);
72     texture.setEnabled(true);
73     texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
74     texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
75     appear.setTexture(texture);
76     return appear;
77 }
78 }

```

程序允许用户通过键盘操纵可视对象——带有纹理的地球仪。窗口底部的文本区域显示了用户通过键盘控制画面的操作说明。

场景图如图10-10所示。同程序清单10-2中一样，生成了一个带有纹理的球体，并附属到一个TransformGroup对象。程序创建一个KeyNavigatorBehavior对象，作用于TransformGroup节点

(第51行)上。行为对象将对用户的输入作出反应,执行相关的变换。KeyNavigatorBehavior对象以一个BoundingSphere对象作为作用范围边界,一个白色的背景对象也引用这个边界对象。

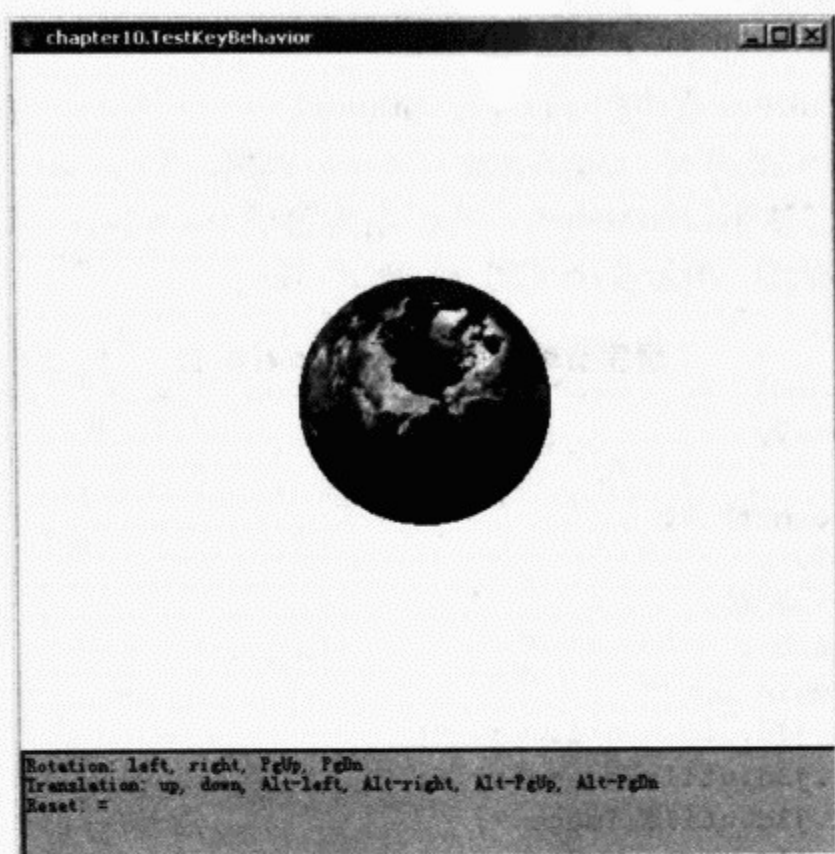


图10-9 通过键盘控制变换

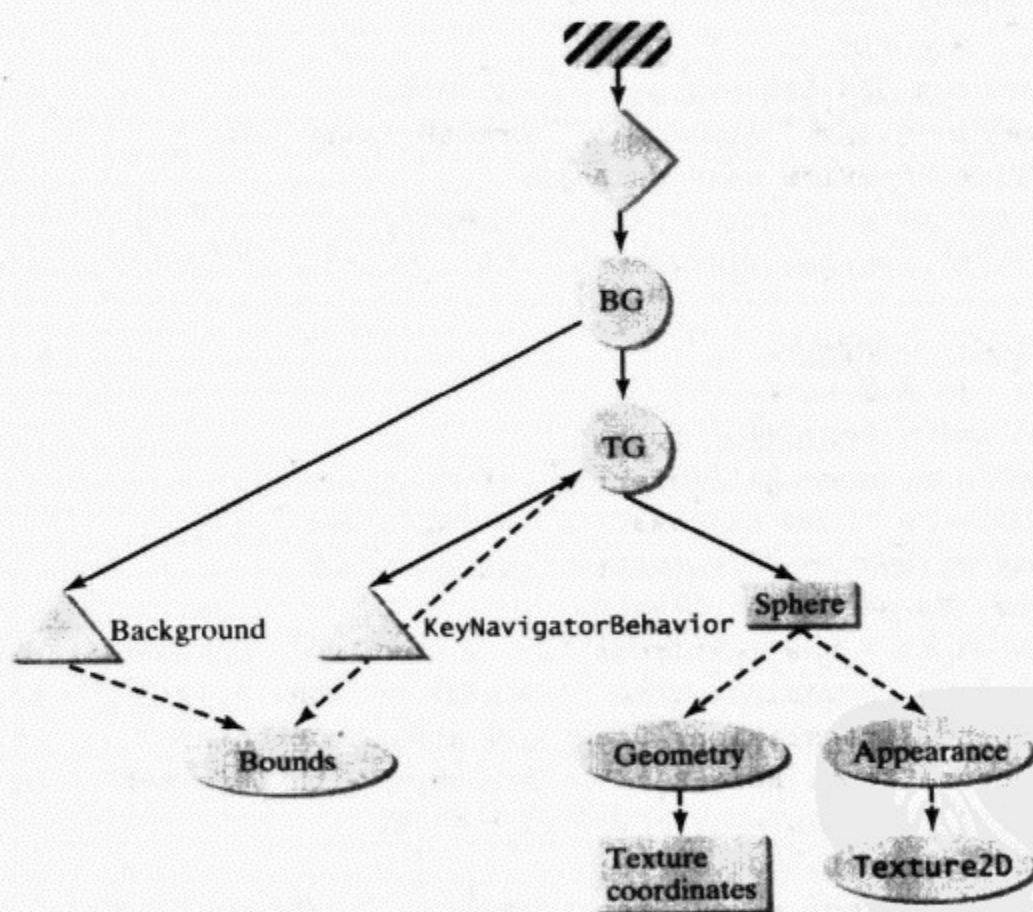


图10-10 键盘行为演示程序的场景图

10.3.3 视图平台行为

ViewPlatformBehavior类支持用户和观察平台之间的交互,OrbitBehavior类允许通过鼠标动作输入改变观察平台。和MouseBehavior类有三个子类支持不同的变换不同,同一个类OrbitBehavior支持了三个操作:转动、平移和缩放。鼠标操作的定义和MouseBehavior类的一样,OrbitBehavior

类使用了一个Canvas3D对象。可以使用下面的构造函数创建一个OrbitBehavior对象：

```
OrbitBehavior ( )
OrbitBehavior (Canvas3D cv)
OrbitBehavior (Canvas3D cv , int flags)
```

通过调用ViewingPlatform类中的setViewPlatformBehavior方法，可以将OrbitBehavior对象加到场景中，其中ViewingPlatform是SimpleUniverse类的组成部分。

程序清单10-5显示了使用OrbitBehavior来控制视图平台，它展示了一个地球，并允许用户通过鼠标动作来转动、平移、缩放整个视图（见图10-11）。

程序清单10-5 MoveView.java

330

```
1 package chapter10;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.net.URL;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.image.*;
11 import com.sun.j3d.utils.behaviors.mouse.*;
12 import com.sun.j3d.utils.behaviors.vp.*;
13 import chapter7.Axes;
14 import java.applet.*;
15 import com.sun.j3d.utils.applet.MainFrame;
16 //定义MoveView类，继承自Applet类，使用OrbitBehavior操纵观察平台
17 public class MoveView extends Applet {
18     public static void main(String[] args) {
19         new MainFrame(new MoveView(), 480, 480); //生成主窗口并设置大小
20     }
21     //重写Applet初始化函数
22     public void init() {
23         GraphicsConfiguration gc =
24             SimpleUniverse.getPreferredConfiguration();
25         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D对象
26         setLayout(new BorderLayout()); //设置布局管理器
27         add(cv, BorderLayout.CENTER);
28         TextArea ta = new TextArea("", 3, 30, TextArea.SCROLLBARS_NONE);
29         ta.setText("Rotation: Drag with left button\n"); //创建并设置文本域
30         ta.append("Translation: Drag with right button\n");
31         ta.append("Zoom: Hold Alt key and drag with left button");
32         ta.setEditable(false); //设置文本域不可编辑
33         add(ta, BorderLayout.SOUTH);
34         BranchGroup root = new BranchGroup(); //创建场景图根节点
35         //创建坐标轴
36         Transform3D tr = new Transform3D();
37         tr.setScale(0.5);
38         tr.setTranslation(new Vector3d(-0.8, -0.7, -0.5));
39         TransformGroup tg = new TransformGroup(tr);
40         root.addChild(tg);
41         Axes axes = new Axes();
42         tg.addChild(axes);
```

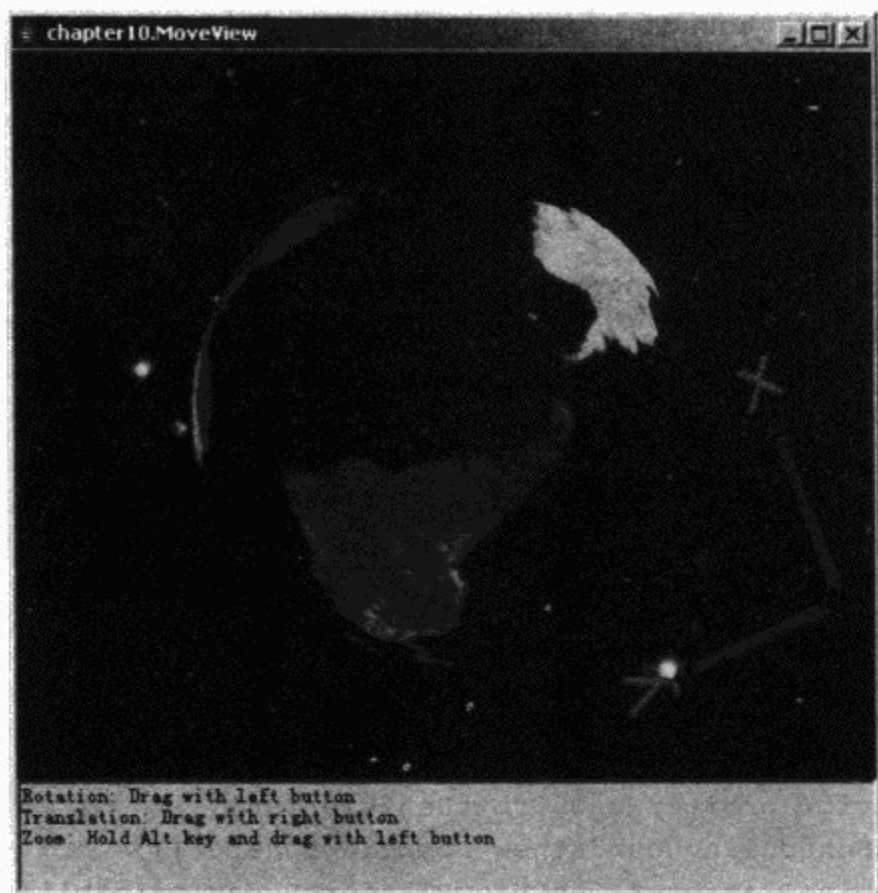


```
43 //创建有纹理的地球仪
44 Appearance ap = createAppearance();
45 root.addChild(new Sphere(0.7f,
46 Primitive.GENERATE_TEXTURE_COORDS, 50, ap));
47 BoundingSphere bounds = new BoundingSphere();
48 //设置光照
49 AmbientLight light = new AmbientLight(true,
50 new Color3f(Color.blue)); //添加蓝色环境光源
51 light.setInfluencingBounds(bounds);
52 root.addChild(light);
53 PointLight ptlight = new PointLight(new Color3f(Color.white),
54 new Point3f(0f, 0f, 2f), new Point3f(1f, 0.3f, 0f)); //添加白色点光源
55 ptlight.setInfluencingBounds(bounds);
56 root.addChild(ptlight);
57 //设置背景
58 Background background = createBackground();
59 background.setApplicationBounds(bounds);
60 root.addChild(background);
61 root.compile();
62 SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
63 su.getViewingPlatform().setNominalViewingTransform();
64 //创建并使用OrbitBehavior对象
65 OrbitBehavior orbit = new OrbitBehavior(cv);
66 orbit.setSchedulingBounds(new BoundingSphere());
67 su.getViewingPlatform().setViewPlatformBehavior(orbit);
68
69 su.addBranchGraph(root);
70 }
71 //创建外观的方法, 使用图像生成纹理
72 Appearance createAppearance(){
73     Appearance appear = new Appearance();
74     URL filename =
75         getClass().getClassLoader().getResource("images/earth.jpg"); //获取图像
76     TextureLoader loader = new TextureLoader(filename, this);
77     Texture texture = loader.getTexture();
78     appear.setTexture(texture);
79     return appear;
80 }
81 //创建背景的方法
82 Background createBackground(){
83     Background background = new Background(); //创建背景对象
84     BranchGroup bg = new BranchGroup();
85     Sphere sphere = new Sphere(1.0f, Sphere.GENERATE_NORMALS | //创建球体
86 Sphere.GENERATE_NORMALS_INWARD |
87 Sphere.GENERATE_TEXTURE_COORDS, 60);
88     Appearance ap = sphere.getAppearance();
89     bg.addChild(sphere);
90     background.setGeometry(bg);
91
92     URL filename =
93         getClass().getClassLoader().getResource("images/stars.jpg"); //获取背景图像
94     TextureLoader loader = new TextureLoader(filename, this); //纹理装载
95     Texture texture = loader.getTexture();
96     ap.setTexture(texture);
```

```

97     return background;
98 }
99 }

```



332

图10-11 控制视图。视图随着鼠标移动而变化

这段程序和程序清单10-3相似，用户可以用鼠标动作转动、平移和缩放整个画面。然而，还有一些不同，因为行为动作是作用于视图平台而不是各个图形对象，所以移动的方向会和程序清单10-3中的方向相反。另一个不同是，在本例中整个场景的视图都在随着鼠标移动而变化，而在程序清单10-3中，只有地球仪是运动的，坐标轴不动。

场景图如图10-12所示。OrbitBehavior作用于与ViewPlatform相关联的变换，此关联结构由以下语句创建：

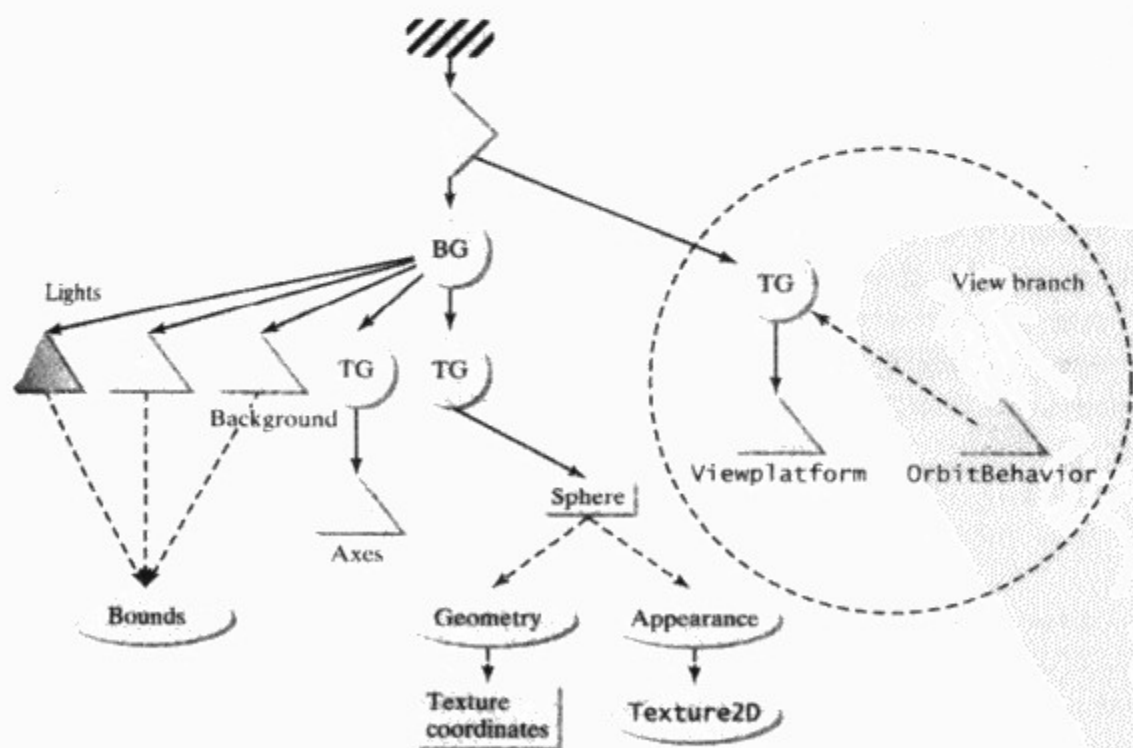


图10-12 视图行为实例的场景图


```
OrbitBehavior orbit = new OrbitBehavior(cv);
Orbit.setSchedulingBounds(new BoundingSphere( ));
Su.getViewingPlatform( ).setViewPlatformBehavior(orbit);
```

因为行为位于视图平台分支 (view branch)，可视对象实际上没有随鼠标动作一起移动，行为改变了视图平台的位置，因此包括地球仪和坐标轴在内的整个场景，看起来好像都在向鼠标操作的相反方向移动。

方法createBackground (第82行) 生成了一个背景，其几何体是带有纹理的球体，背景使得我们能更清楚地观察到视图平台的变动。

10.4 行为和拾取

通常，将拾取和行为联合起来应用，以完成与特定可视对象的交互。拾取功能提供了动态选择可视对象的机制，行为结构提供了系统化的向场景中引入操纵的方式。

333

10.4.1 拾取和鼠标行为

包括PickRotateBehavior、PickTranslateBehavior与PickZoomBehavior等在内的PickMouseBehavior类将拾取和鼠标行为结合起来。鼠标控制的转动、平移和缩放操作和MouseBehavior中定义的一样，只应用于被选中的物体。创建PickMouseBehavior节点涉及Canvas3D对象、用于拾取的场景图分支的根以及作用范围边界。例如，下面的代码段在根节点上设立了PickRotateBehavior：

```
PickRotateBehavior rotator = new PickRotateBehavior(cv , root , bounds);
root.addChild(rotator);
```

必须在待选择的对象上放一个TransformGroup节点，只有这样，行为才能有作用的对象。还需要将选中对象和变换节点的相应能力比特设置为有效。

程序清单10-6给出了PickMouseBehavior类的应用，程序显示的画面中，有一组随机放置的海鸥（见图10-13），用户可以通过移动鼠标独立地转动、平移和缩放其中任何一个海鸥。

程序清单10-6 TestPickBehavior.java

```
1 package chapter10;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.net.URL;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.picking.PickTool;
11 import com.sun.j3d.utils.picking.behaviors.*;
12 import chapter7.Axes;
13 import java.applet.*;
14 import com.sun.j3d.utils.applet.MainFrame;
15 //定义TestPickBehavior类，继承自Applet类，演示PickMouseBehavior的使用
16 public class TestPickBehavior extends Applet {
17     public static void main(String[] args) {
18         new MainFrame(new TestPickBehavior(), 480, 480); //创建主窗口并设置大小
19     }
20     //重写Applet的初始化函数
```

```

21 public void init() {
22     GraphicsConfiguration gc =
23         SimpleUniverse.getPreferredConfiguration();
24     Canvas3D cv = new Canvas3D(gc); //创建Canvas3D对象
25     setLayout(new BorderLayout()); // 布局管理器
26     add(cv, BorderLayout.CENTER); //画板位置居中
27     TextArea ta = new TextArea("", 3, 30, TextArea.SCROLLBARS_NONE);
28     ta.setText("Rotation: Drag with left button\n"); //创建并设置文本域
29     ta.append("Translation: Drag with right button\n");
30     ta.append("Zoom: Hold Alt key and drag with left button");
31     ta.setEditable(false); //设置文本域不可编辑
32     add(ta, BorderLayout.SOUTH);
33     BranchGroup bg = createSceneGraph(cv); //创建场景图
34     bg.compile();
35     SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
36     su.getViewingPlatform().setNominalViewingTransform();
37     su.addBranchGraph(bg);
38 }
39 //生成BranchGroup的私有方法, 创建场景图
40 private BranchGroup createSceneGraph(Canvas3D cv) {
41     BranchGroup root = new BranchGroup();
42     //加入8个海鸥对象
43     for (int i = 0; i < 8; i++)
44         root.addChild(createObject());
45
46     BoundingSphere bounds = new BoundingSphere();
47     //设置拾取旋转行为
48     PickRotateBehavior rotator =
49         new PickRotateBehavior(root, cv, bounds,
50             PickTool.GEOMETRY);
51     root.addChild(rotator);
52     //设置拾取平移行为
53     PickTranslateBehavior translator =
54         new PickTranslateBehavior(root, cv,
55             bounds, PickTool.GEOMETRY);
56     root.addChild(translator);
57     //设置拾取缩放行为
58     PickZoomBehavior zoom = new PickZoomBehavior(root, cv, bounds,
59         PickTool.GEOMETRY);
60     root.addChild(zoom);
61     //设置光照
62     AmbientLight light = new AmbientLight(true,
63         new Color3f(Color.blue)); //添加蓝色环境光源
64     light.setInfluencingBounds(bounds);
65     root.addChild(light);
66     PointLight ptlight = new PointLight(new Color3f(Color.white),
67         new Point3f(0f, 0f, 2f), new Point3f(1f, 0.3f, 0f)); //添加白色点光源
68     ptlight.setInfluencingBounds(bounds);
69     root.addChild(ptlight);
70     //设置背景
71     Background background = new Background(1.0f, 1.0f, 1.0f); //设置背景颜色
72     background.setApplicationBounds(bounds);
73     root.addChild(background);
74     return root;

```

334


```
75 }
76 //生成海鸥对象节点方法
77 private Node createObject() {
78     //设置变换
79     Transform3D trans = new Transform3D();
80     trans.setTranslation(new Vector3d(Math.random()-0.5,
81         Math.random()-0.5,Math.random()-0.5));
82     trans.setScale(0.3);
83     TransformGroup spin = new TransformGroup(trans); //设置变换属性
84     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
85     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
86     spin.setCapability(TransformGroup.ENABLE_PICK_REPORTING);
87     //创建可视形体
88     Appearance ap = new Appearance();
89     ap.setMaterial(new Material());
90     Shape3D shape = new Shape3D(new GullCG(), ap); //生成海鸥形体
91     PickTool.setCapabilities(shape, PickTool.INTERSECT_FULL);
92     spin.addChild(shape);
93     return spin;
94 }
95 }
```

335

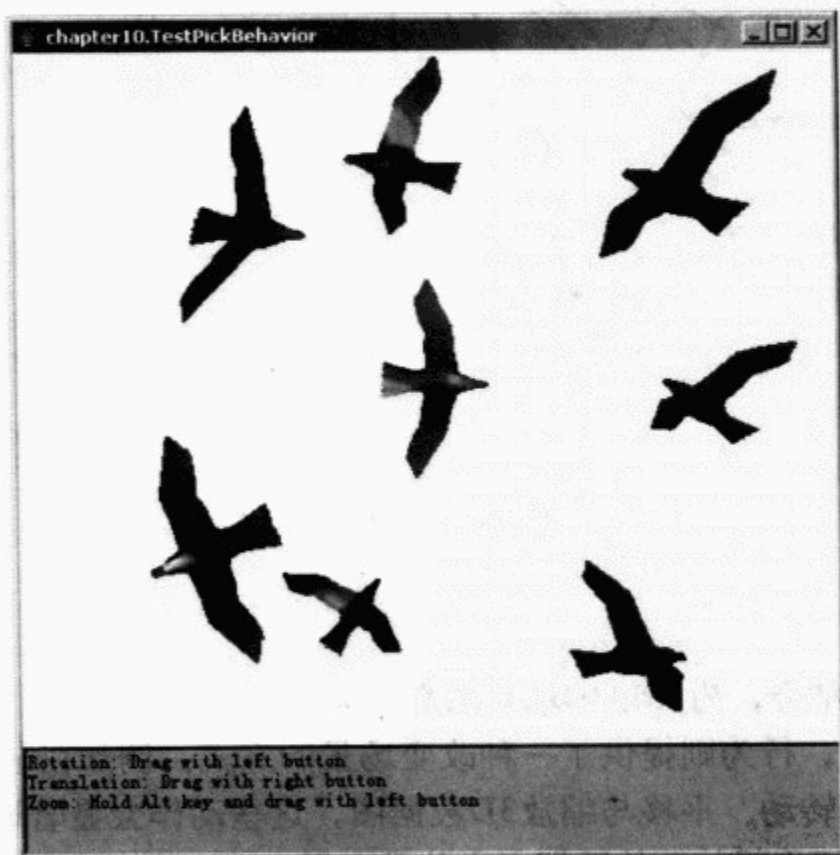


图10-13 PickMouseBehavior类允许对单个对象独立地进行操作

画面中包含八只随机放置的海鸥，每一只海鸥都可以用鼠标选中进行操作，按着鼠标左键拖动鼠标使海鸥转动，按住右键拖动鼠标使海鸥平移，按着鼠标中键拖动鼠标（或在左键按下时按住Alt键）缩放海鸥。

场景图如图10-14所示。创建了三个PickMouseBehavior对象作用于BranchGroup根节点上（第47~60行），这三个对象分别是PickRotateBehavior对象、PickTranslateBehavior对象与PickZoomBehavior对象。八只海鸥有同样的结构。createObject方法创建了场景图的一个分支，它包含一个TransformGroup对象和一个Shape3D对象。GullCG类定义了Shape3D节点的几何属性，该形状应用了带有Material组件的Appearance对象，独立地针对每一个海鸥进行拾取—鼠标

行为，需要每个形状有一个专用的TransformGroup节点。在TransformGroup节点上，设置了以下能力比特（第85~87行）：

```
TransformGroup.ALLOW_TRANSFORM_WRITE
TransformGroup.ALLOW_TRANSFORM_READ
TransformGroup.ENABLE_PICK_REPORTING
```

对每一个形状，调用下面的方法设置所有相关的能力比特（第92行）：

```
PickTool.setCapabilities(shape , PickTool.INTERSECT_FULL);
```

场景中加入两个光源，对具有带材质属性的外观的海鸥进行光照。在场景中，还加入了一个白色的背景，所有的行为节点、光源和背景节点都共享同样的边界对象。

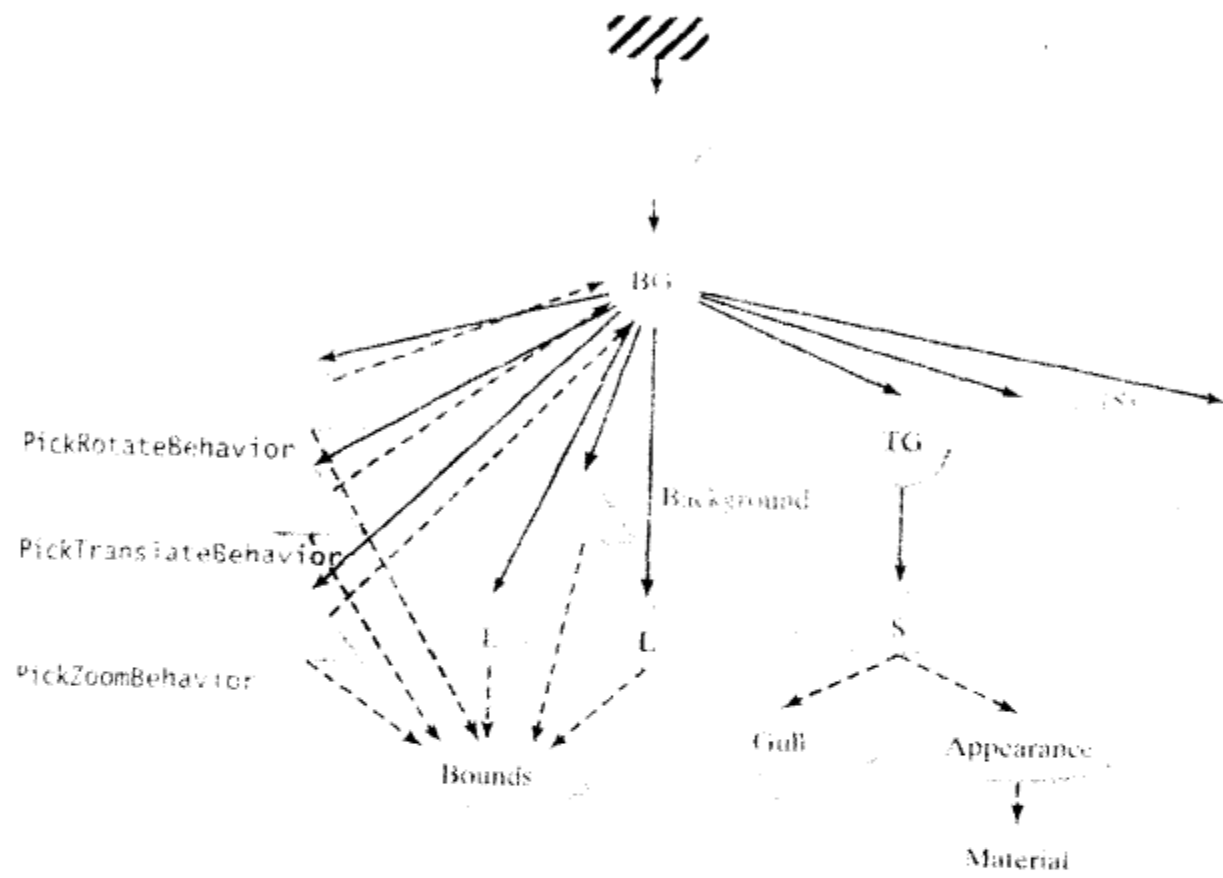


图10-14 场景图

10.4.2 数据可视化

336

拾取功能和行为的结合，为人和3D场景的交互提供了强大的工具。拾取功能使得用户可以选择特定的场景图对象，行为则提供了一种改变场景的方式。例如在一个数据可视化应用中，它可以帮助用户用鼠标转动、平移与缩放3D数据图，这些动作会显著地改善人们对3D结构的感知效果。同时，用鼠标选择特殊的数据点也非常有用。

程序清单10-7给出了用拾取和行为完成这些功能的方法，这段程序显示了一些分散的3D点（见图10-15），3D数据点用有色的圆点表示，用户可以转动、平移和缩放这些点，以便从不同的视角观察这些数据，这些动作将帮助用户更好地在2D的显示屏上理解3D结构的数据。用户还可以通过点击鼠标单独选中这些点，窗口底部的文本区显示了选择的点的索引和坐标。

程序清单10-7 DataView.java

```
1 package chapter10;
2
3 import javax.vecmath.*;
4 import java.awt.*;
```



```
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import com.sun.j3d.utils.picking.*;
10 import com.sun.j3d.utils.behaviors.mouse.*;
11 import chapter7.Axes;
12 import java.applet.*;
13 import com.sun.j3d.utils.applet.MainFrame;
14 //定义DataViewer3D类, 继承自Applet类
15 public class DataViewer3D extends Applet {
16     public static void main(String[] args) {
17         new MainFrame(new DataViewer3D(), 640, 480); //创建主窗口并设置大小
18     }
19
20     PointArray geom; //声明存放点的点数组变量
21     PickCanvas pc; //声明用于拾取的画板变量
22     TextField text; // 文本区
23     //重写Applet初始化函数
24     public void init() {
25         setLayout(new BorderLayout()); //设置布局管理器
26         GraphicsConfiguration gc =
27             SimpleUniverse.getPreferredConfiguration();
28         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D对象
29         add(cv, BorderLayout.CENTER);
30         cv.addMouseListener(new MouseAdapter() { //加入鼠标事件侦听器
31             public void mouseClicked
32                 (java.awt.event.MouseEvent mouseEvent) { //响应鼠标点击事件
33                 pick(mouseEvent); //拾取操作
34             }
35         });
36         text = new TextField(); //初始化文本字段
37         add(text, BorderLayout.SOUTH);
38         BranchGroup bg = createSceneGraph(cv); //创建场景图
39         bg.compile();
40         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
41         su.getViewingPlatform().setNominalViewingTransform();
42         su.addBranchGraph(bg); // 将节点加入场景图
43     }
44     //生成BranchGroup的私有方法, 用于生成场景图
45     private BranchGroup createSceneGraph(Canvas3D cv) {
46         BranchGroup root = new BranchGroup();
47         TransformGroup spin = new TransformGroup();
48         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
49         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
50         root.addChild(spin);
51         //创建坐标轴
52         Transform3D tr = new Transform3D();
53         tr.setScale(0.3);
54         TransformGroup tg = new TransformGroup(tr);
55         spin.addChild(tg);
56         Axes axes = new Axes();
57         tg.addChild(axes);
58         //设置外观
```

337

```

59 Appearance ap = new Appearance();
60 ap.setPointAttributes(new PointAttributes(10f, true));
61 //随机生成20个数据点
62 int n = 20;
63 geom = new PointArray(n,
64     PointArray.COORDINATES | PointArray.COLOR_4); //定义点数组
65 geom.setCapability(PointArray.ALLOW_COORDINATE_READ); //设置能力比特
66 geom.setCapability(PointArray.ALLOW_FORMAT_READ);
67 geom.setCapability(PointArray.ALLOW_COLOR_READ);
68 geom.setCapability(PointArray.ALLOW_COLOR_WRITE);
69 geom.setCapability(PointArray.ALLOW_COUNT_READ);
70 Point3f[] coords = new Point3f[n];
71 Color4f[] colors = new Color4f[n];
72 for (int i = 0; i < n; i++) { //随机生成数据点
73     coords[i] = new Point3f((float)(Math.random()-0.5),
74         (float)(Math.random()-0.5), (float)(Math.random()-0.5));
75     colors[i] = new Color4f((float)(Math.random()),
76         (float)(Math.random()), (float)(Math.random()), 1f);
77 }
78 geom.setCoordinates(0, coords); //设置geom的点坐标数组
79 geom.setColors(0, colors); //设置geom的颜色坐标数组
80 BranchGroup bg = new BranchGroup(); //创建BranchGroup对象
81 spin.addChild(bg);
82 pc = new PickCanvas(cv, bg); //初始化PickCanvas对象
83 pc.setTolerance(5);
84 pc.setMode(PickTool.GEOMETRY_INTERSECT_INFO); //设置拾取模式
85 Shape3D shape = new Shape3D(geom, ap); //创建几何形体
86 bg.addChild(shape);
87 PickTool.setCapabilities(shape, PickTool.INTERSECT_TEST);
88 shape.setCapability(Shape3D.ALLOW_GEOMETRY_READ);
89 //设置鼠标旋转行为
90 MouseRotate rotator = new MouseRotate(spin);
91 BoundingSphere bounds = new BoundingSphere();
92 rotator.setSchedulingBounds(bounds);
93 spin.addChild(rotator);
94 //设置鼠标平移行为
95 MouseTranslate translator = new MouseTranslate(spin);
96 translator.setSchedulingBounds(bounds);
97 spin.addChild(translator);
98 //设置鼠标缩放行为
99 MouseZoom zoom = new MouseZoom(spin);
100 zoom.setSchedulingBounds(bounds);
101 spin.addChild(zoom);
102 //设置背景和光照
103 Background background = new Background(1.0f, 1.0f, 1.0f); //设定背景颜色
104 background.setApplicationBounds(bounds);
105 root.addChild(background);
106 AmbientLight light =
107     new AmbientLight(true, new Color3f(Color.red)); //添加红色环境光源
108 light.setInfluencingBounds(bounds);
109 root.addChild(light);
110 PointLight ptlight = new PointLight(new Color3f(Color.green),
111     new Point3f(3f, 3f, 3f), new Point3f(1f, 0f, 0f)); //添加绿色点光源
112 ptlight.setInfluencingBounds(bounds);

```



```

113     root.addChild(ptlight);
114     PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
115     new Point3f(-2f,2f,2f), new Point3f(1f,0f,0f)); //添加橙色点光源
116     ptlight2.setInfluencingBounds(bounds);
117     root.addChild(ptlight2);
118     return root;
119 }
120 //拾取操作的处理函数
121 private void pick(MouseEvent mouseEvent) {
122     Color4f color = new Color4f();
123     pc.setShapeLocation(mouseEvent);
124     PickResult[] results = pc.pickAll();
125     for (int i = 0; (results != null) &&
126         (i < results.length); i++) {
127         PickIntersection inter = results[i].getIntersection(0); //得到交点
128         Point3d pt = inter.getClosestVertexCoordinates();
129         int[] ind = inter.getPrimitiveCoordinateIndices(); //得到被选择点的坐标
130         text.setText("vertex " + ind[0] + ": (" + pt.x + ", "
131             + pt.y + ", " + pt.z + ")"); //文本字段显示索引和点的坐标
132         geom.setColor(ind[0], color); //颜色调整
133         color.x = 1f - color.x;
134         color.y = 1f - color.y;
135         color.z = 1f - color.z;
136         if (color.w > 0.8) color.w = 0.5f;
137         else color.w = 1f;
138         geom.setColor(ind[0], color);
139     }
140 }
141 }

```

339

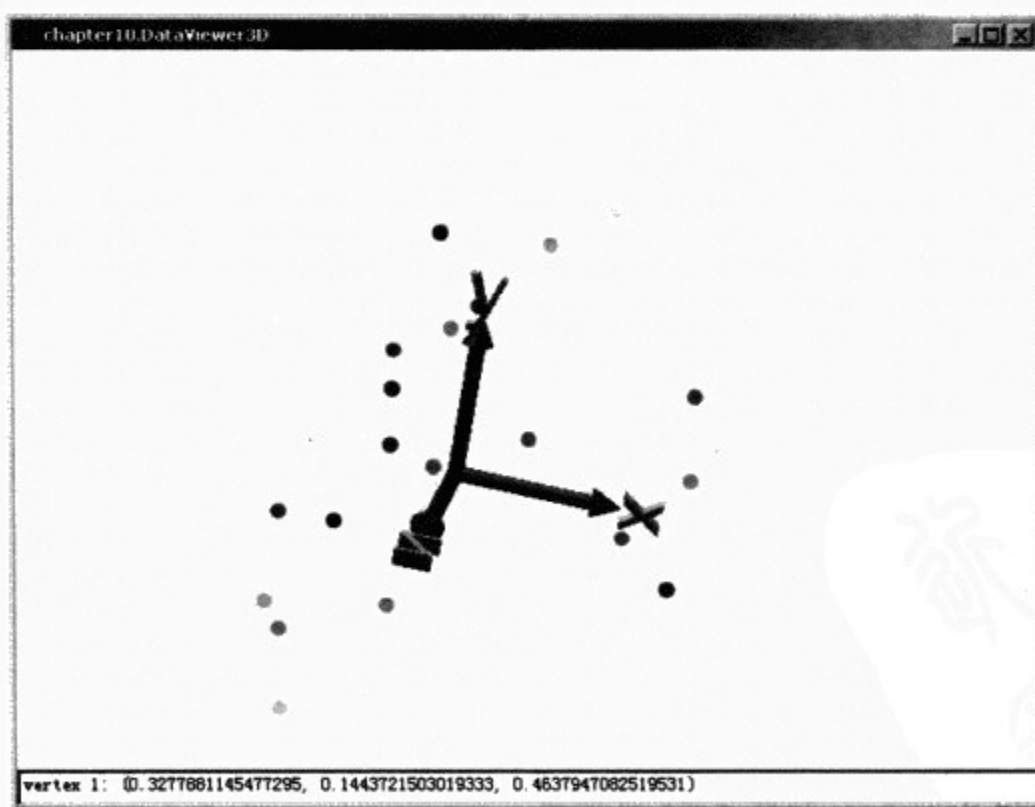


图10-15 用户控制下的3D分散数据点的选择和转动

程序的场景图如图10-16所示。设置MouseRotate对象、MouseTranslate对象和MouseZoom对象等三个MouseBehavior对象，用于操作一个TransformGroup节点。这些行为使得用户可以转动、平移与缩放这些点。

待绘制的数据点是随机地产生的，并用一个PointArray表示（第61~79行）。点的颜色是用Color4f对象随机分配的，一个Axes对象用来提供视觉参考的坐标系。一个特定的BranchGroup节点放置在数据对象的上面，它是用于拾取操作的。拾取不是作用于根BranchGroup节点，因为Axes对象将排除在拾取范围之外。

340 一个鼠标事件监听器放置在Canvas3D对象之上，它监听鼠标的点击事件，并通过调用pick方法（第121行）实现对数据点分支的拾取。一个PickCanvas对象用于实现基于鼠标位置的拾取，因为所有的数据点都属于一个对象的顶点，所以需要得到拾取的交叉点（intersection），以确定特定的点（第127~128行）：

```
PickIntersection inter = results[i].getIntersection(0);
Point3D pt = inter.getClosestVertexCoordinates();
```

还可以得到几何体的顶点的索引：

```
int[] ind = inter.getPrimitiveCoordinateIndices();
```

当得到交叉点的索引时，在窗口底部的文本区中将显示这个点的索引和坐标。被选中的点将以不同的颜色进行显示，新的颜色是原始颜色的补色，这样一来，如果两次选中同一个点，那么这个点的颜色将变回原始的颜色。

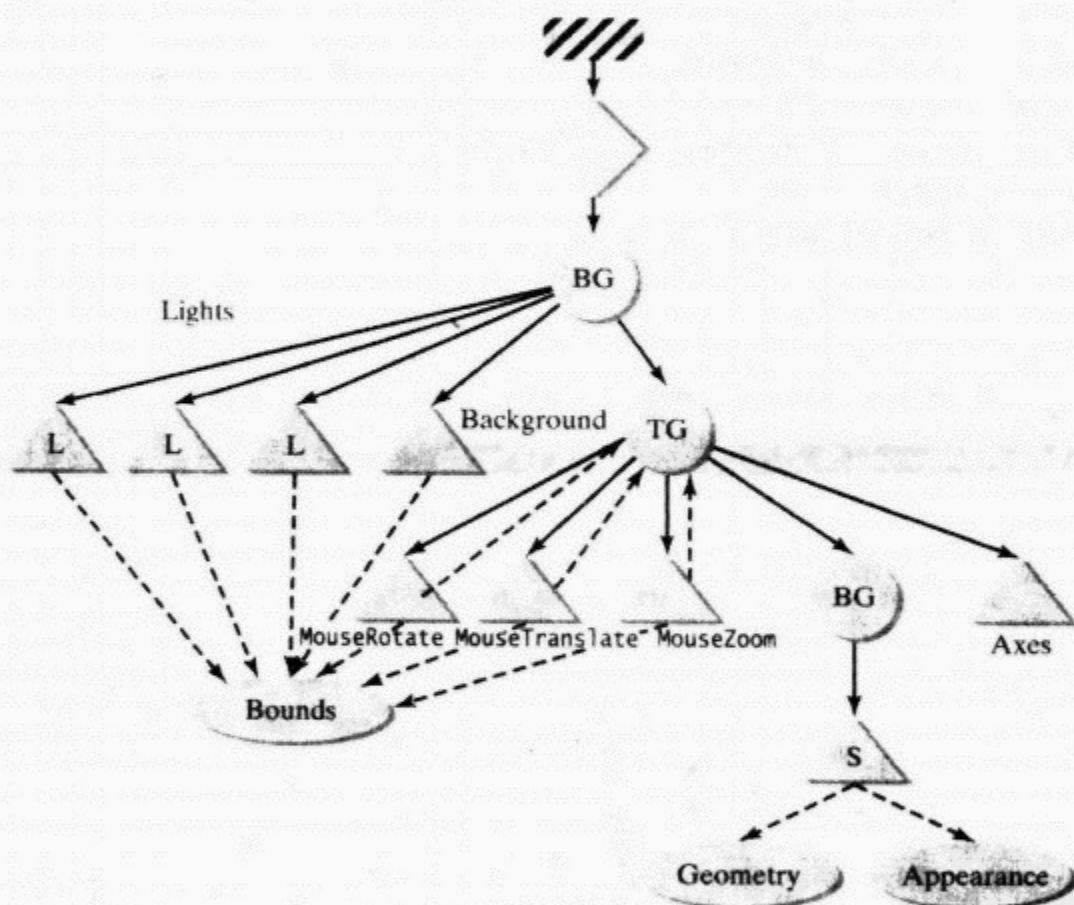


图10-16 场景图

主要的类和方法

- javax.media.j3d.Behavior 封装动态行为的类。
- javax.media.j3d.WakeupCondition 唤醒条件的基类。
- javax.media.j3d.WakeupCriterion 封装各种特定唤醒条件的基类。
- javax.media.j3d.WakeupOnElapsedTime 基于流逝的时间的唤醒标准。
- javax.media.j3d.WakeupOnAWTEvent 基于AWT事件的唤醒标准。

- `javax.media.j3d.WakeupAnd` 用逻辑与 (AND) 合并唤醒标准的类。
- `javax.media.j3d.WakeupOr` 用逻辑或 (OR) 合并唤醒标准的类。
- `javax.media.j3d.WakeupAndOfOrs` 用逻辑与 (AND) 合并 `WakeupOr` 唤醒条件的类。
- `javax.media.j3d.WakeupOrOfAnds` 用逻辑或 (OR) 合并 `WakeupAnd` 唤醒条件的类。
- `com.sun.j3d.utils.behaviors.mouse.MouseBehavior` 用于鼠标驱动行为的基类。
- `com.sun.j3d.utils.behaviors.mouse.MouseRotate` 鼠标控制的旋转行为。
- `com.sun.j3d.utils.behaviors.mouse.MouseTranslate` 鼠标控制的平移行为。
- `com.sun.j3d.utils.behaviors.mouse.MouseZoom` 鼠标控制的缩放行为。
- `com.sun.j3d.utils.behaviors.keyboard.KeyNavigatorBehavior` 基于键盘的变换行为。
- `com.sun.j3d.utils.behaviors.vp.OrbitBehavior` 鼠标驱动的视图平台变换行为。
- `com.sun.j3d.utils.picking.behaviors.PickMouseBehavior` 鼠标驱动的拾取和变换行为基类。
- `com.sun.j3d.utils.picking.behaviors.PickRotateBehavior` 鼠标控制的拾取和旋转行为。
- `com.sun.j3d.utils.picking.behaviors.PickTranslateBehavior` 鼠标控制的拾取和平移行为。
- `com.sun.j3d.utils.picking.behaviors.PickZoomBehavior` 鼠标控制的拾取和缩放行为。
- `javax.media.j3d.MouseBehavior.initialize()` 由调度程序 (scheduler) 调用一次的初始化方法。
- `javax.media.j3d.MouseBehavior.processStimulus(Enumeration)` 唤醒条件发生时调用的方法。
- `javax.media.j3d.MouseBehavior.wakeupOn(WakeupCondition)` 设置唤醒条件的方法。

关键术语

- 行为 (behavior) 3D场景中的动态模型。
- 交互 (interaction) 与用户输入相关的动态行为。
- 唤醒条件 (wakeup condition) 触发行为的条件。
- 拾取行为 (picking behavior) 基于拾取的行为。

本章提要

- 本章介绍了行为的概念，行为系统地表示了3D场景中的动态效果。通常使用到的动态变化，例如交互和动画，都可以用行为来模拟。本章讨论了一般的行为机制和交互行为，动画将在下一章中介绍。
- Java 3D提供了一组Behavior和WakeupCondition类，来实现各种行为操作。Behavior对象建立一个唤醒条件，并对这个唤醒条件作出响应，以完成它们的任务。Java 3D通过WakeupCriterion类层次提供了大量的特定唤醒条件，同时，可以通过逻辑运算组合这些唤醒标准，产生复合唤醒条件。
- 交互通常由行为及与鼠标、键盘动作等用户输入事件相关的唤醒条件实现。Java 3D包含几个工具包用来支持交互。
- MouseBehavior族类提供了通过鼠标动作控制TransformGroup节点的行为，Key-NavigatorBehavior类使用键盘来控制变换，ViewPlatformBehavior类对视图平台进行操作。
- 拾取通常和行为结合起来，用于构造交互。PickMouseBehavior类族中的工具类，可以便捷地构造那些把拾取和仿射变换关联起来的行为。

342

复习题

- 10.1 讨论将行为定义成场景图节点的优点。
- 10.2 描述交互和动画之间的区别。
- 10.3 哪些WakeupCriterion类通常和交互相关联？

10.4 描述MouseBehavior类和ViewPlatformAWTBehavior类之间的区别。

10.5 在程序清单10-6中，如果所有海鸥与同一个TransformGroup节点相关，而不是与各自的变换相关，会怎样？

编程练习

10.1 写一个Java 3D程序，用3D文本显示实时的数字式时钟。

10.2 在题10.1的程序中加入MouseRotate行为，使得可以通过拖动鼠标来旋转时钟。

10.3 在题10.2的程序中加入MouseTranslate行为，使得可以通过移动鼠标来平移时钟。

10.4 在题10.3的程序中加入MouseZoom行为，使得可以通过拖动鼠标来缩放时钟。

343 10.5 在题10.4的程序中加入KeyNavigationBehavior行为，使得可以通过键盘来操纵时钟。

10.6 修改程序清单10-6的代码，使得只使用一个GullCG几何体实例。

10.7 重写程序清单10-7，使用OrbitBehavior，通过鼠标动作来控制视图。

344 10.8 重写程序清单10-7，用一个Behavior代替鼠标事件监听器，来处理拾取。

第11章 动 画

学习目标

- 理解动画的概念和方法。
- 理解和创建Alpha类对象。
- 用不同的插值器生成动画。
- 用Morph节点和行为创建变形效果。
- 使用LOD行为。
- 使用公告板行为。

345

11.1 引言

动画是随时间变化的、具有逼真运动效果的图形绘制序列。虽然交互和动画在图形系统中都产生动态行为，但是动画通常是与时间相关的，并且是与帧相关的。帧是反映变化过程的中间结果，它是动画的主体。

在计算机图形学中，动画常用来产生动态效果。为了产生动画图形效果，绘制场景必须随时间变化，这些变化可能包括绘制场景的不同属性，如几何特性、变换、位置、颜色和透明度等。

Java 3D通过Behavior类为在场景图中生成动画提供扩展支持。Behavior类族中的插值器(interpolator)封装了生成一般动画的基本功能。Alpha类实现了驱动插值器的时间函数，它能够通过插值器简单地生成动画，用户要做的，只是创建带Alpha值的适当的插值器类对象，并把它链接到目标对象上。

其他与动画相关的技术包括：变形、细节层次(level of details, LOD)和公告板行为(billboard behavior)。变形对几何体间的变化进行平滑处理。LOD提供了根据对象与眼睛之间的距离，在不同层次上绘制细节的方法。公告板行为提供了自动调整对象方向的便捷方法，使该对象始终面向观察者。

11.2 Alpha对象

动画通过时间实现场景的动态变化。一个Alpha对象定义一个时间函数（一般是周期性的），生成0.0到1.0之间的值。使用Alpha对象驱动动画，比使用实时时钟更方便。Java3D的Alpha类是NodeComponnet类的一个子类。Alpha对象为动画工具——插值器提供了输入。Alpha对象的功能的形状由几个参数确定。Alpha对象的波形如图11-1所示。

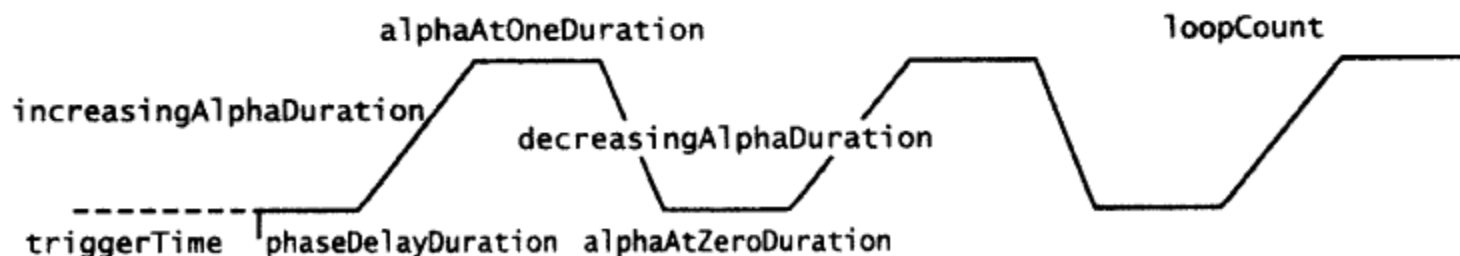


图11-1 Alpha对象的波形

Java 3D的Alpha对象包括如下参数：

- **LoopCount**: 脉冲或周期变化的次数，-1表示无限循环。
- **startTime**: 初始化参考点的绝对时间。
- **triggerTime**: 从starTime到激活Alpha对象的时间，一般在微秒级。
- **phaseDelayDuration**: 从triggerTime到初始化延时的时间，一般在微秒级。
- **alphaAtZeroDuration**: alpha对象值保持在0.0的时间，一般在微秒级。
- **alphaAtOneDuration**: alpha对象值保持在1.0的时间，一般在微秒级。
- **increasingAlphaDuration**: alpha对象值从0.0增长到1.0的时间，一般在微秒级。
- **decreasingAlphaDuration**: alpha对象值从1.0减少到0.0的时间，一般在微秒级。
- **increasingAlphaRampDuration**: alpha对象值在增长阶段累加的时间，一般在微秒级。
- **decreasingAlphaRampDuration**: alpha对象值在减少阶段累加的时间，一般在微秒级。

346

increasingAlphaRampDuration参数和decreasingAlphaRampDuration参数定义了波形上的某种平滑效果，increasingAlphaRampDuration参数的值指明了在增长阶段开始时加速的时间间隔和在增长阶段结束时减速的时间间隔。类似地，decreasingAlphaRampDuration参数指明了在减少阶段加速和减速的时间间隔。

下面介绍Alpha对象可以利用的构造函数：

```
public Alpha()
public Alpha(int loopCount, long increasingAlphaDuration)
public Alpha(int loopCount, long triggerTime, long phaseDelayDuration, long
    increasingAlphaDuration, long increasingAlphaRampDuration, long
    alphaAtOneDuration )
public Alpha(int loopCount, int mode, long triggerTime,
    long phaseDelayDuration, long increasingAlphaDuration,
    long increasingAlphaRampDuration, long alphaAtOneDuration,
    long decreasingAlphaDuration,
    long decreasingAlphaRampDuration, long alphaAtZeroDuration )
```

Alpha对象的alpha值可由如下value方法获得：

```
public float value()
public float value(long atTime)
```

第一个方法得到当前alpha的值，第二个方法得到指定时间alpha的值。

程序清单11-1绘制了Alpha对象的波形（如图11-2所示）。用户能在运行时改变Alpha对象的参数，以便展示波形变化的效果。

程序清单11-1 TestAlpha.java

```
1 package chapter11;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import javax.media.j3d.*;
7 定义TestAlpha类，继承自JApplet，演示Alpha对象的波形图
8 public class TestAlpha extends JApplet {
9     public static void main(String s[]) {
10         JFrame frame = new JFrame();//创建主窗口
11         frame.setTitle("Alpha");//设置主窗口名称
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         JApplet applet = new TestAlpha();//创建TestAlpha对象
```



```
14     applet.init();//初始化TestAlpha对象
15     frame.getContentPane().add(applet);//TestAlpha对象加入主窗口
16     frame.pack();
17     frame.setVisible(true);//设置主窗口可见
18 }
19 //声明并创建Alpha类对象变量
20 Alpha alpha = new Alpha();
21 Plot plot;//声明波形图绘制组件变量
22 JTextField tfLoopCount;//声明数据输入文本字段
23 JTextField tfTriggerTime;
24 JTextField tfAlphaAtZeroDuration;
25 JTextField tfAlphaAtOneDuration;
26 JTextField tfIncreasingAlphaDuration;
27 JTextField tfDecreasingAlphaDuration;
28 JTextField tfIncreasingAlphaRampDuration;
29 JTextField tfDecreasingAlphaRampDuration;
30 //重写Applet的初始化函数
31 public void init() {
32     Container cp = this.getContentPane();
33     cp.setLayout(new BorderLayout());//设置布局管理器
34     plot = new Plot();//初始化Plot对象
35     cp.add(plot, BorderLayout.CENTER);
36     JPanel p = new JPanel();
37     p.setBorder(BorderFactory.createTitledBorder(
38         "Alpha parameters"));
39     cp.add(p, BorderLayout.SOUTH);
40     p.setLayout(new GridLayout(5, 4, 10, 5));
41     p.add(new JLabel("loopCount"));//加入数据录入的标签及文本字段
42     tfLoopCount = new JTextField("-1");
43     p.add(tfLoopCount);
44     p.add(new JLabel("triggerTime"));
45     tfTriggerTime = new JTextField("0");
46     p.add(tfTriggerTime);
47     p.add(new JLabel("alphaAtZeroDuration"));
48     tfAlphaAtZeroDuration = new JTextField("0");
49     p.add(tfAlphaAtZeroDuration);
50     p.add(new JLabel("alphaAtOneDuration"));
51     tfAlphaAtOneDuration = new JTextField("0");
52     p.add(tfAlphaAtOneDuration);
53     p.add(new JLabel("increasingAlphaDuration"));
54     tfIncreasingAlphaDuration = new JTextField("1000");
55     p.add(tfIncreasingAlphaDuration);
56     p.add(new JLabel("decreasingAlphaDuration"));
57     tfDecreasingAlphaDuration = new JTextField("0");
58     p.add(tfDecreasingAlphaDuration);
59     p.add(new JLabel("increasingAlphaRampDuration"));
60     tfIncreasingAlphaRampDuration = new JTextField("0");
61     p.add(tfIncreasingAlphaRampDuration);
62     p.add(new JLabel("decreasingAlphaRampDuration"));
63     tfDecreasingAlphaRampDuration = new JTextField("0");
64     p.add(tfDecreasingAlphaRampDuration);
65     p.add(new JPanel());
66     JButton button = new JButton("Plot");//创建Plot按钮, 添加按钮的事件侦听器
67     p.add(button);
```

```

68     button.addActionListener(new ActionListener() {
69         public void actionPerformed(ActionEvent ev) {
70             setAlpha();//重新设置Alpha对象的参数
71             repaint();//刷新
72         }
73     });
74     p.add(new JPanel());
75     button = new JButton("Reset");//创建Reset按钮, 添加按钮的事件侦听器
76     p.add(button);
77     button.addActionListener(new ActionListener() {
78         public void actionPerformed(ActionEvent ev) {
79             tfLoopCount.setText("-1");//重新设置各文本字段
80             tfTriggerTime.setText("0");
81             tfAlphaAtZeroDuration.setText("0");
82             tfAlphaAtOneDuration.setText("0");
83             tfIncreasingAlphaDuration.setText("1000");
84             tfDecreasingAlphaDuration.setText("0");
85             tfIncreasingAlphaRampDuration.setText("0");
86             tfDecreasingAlphaRampDuration.setText("0");
87             setAlpha();//重新设置Alpha对象的参数
88             repaint();//刷新
89         }
90     });
91 }
92 //根据用户的输入值, 对Alpha对象的参数重新设置
93 void setAlpha() {
94     alpha.setMode
95         (Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE);
96     int n = Integer.parseInt(tfLoopCount.getText());//获取LoopCount值
97     alpha.setLoopCount(n);
98     n = Integer.parseInt(tfTriggerTime.getText());
99     alpha.setTriggerTime(n);
100    n = Integer.parseInt(tfAlphaAtZeroDuration.getText());
101    alpha.setAlphaAtZeroDuration(n);
102    n = Integer.parseInt(tfAlphaAtOneDuration.getText());
103    alpha.setAlphaAtOneDuration(n);
104    n = Integer.parseInt(tfIncreasingAlphaDuration.getText());
105    alpha.setIncreasingAlphaDuration(n);
106    n = Integer.parseInt(tfDecreasingAlphaDuration.getText());
107    alpha.setDecreasingAlphaDuration(n);
108    n = Integer.parseInt(tfIncreasingAlphaRampDuration.getText());
109    alpha.setIncreasingAlphaRampDuration(n);
110    n = Integer.parseInt(tfDecreasingAlphaRampDuration.getText());
111    alpha.setDecreasingAlphaRampDuration(n);
112 }
113 //定义用于显示波形图的Plot类, 继承自JPanel
114 class Plot extends JPanel {
115     public Plot() { //构造函数, 设置JPanel的背景、大小等属性
116         this.setBackground(Color.white);
117         this.setBorder(BorderFactory.createLoweredBevelBorder());
118         this.setPreferredSize(new Dimension(800, 200));
119     }
120     //重写组件绘制函数, 绘制波形
121     public void paintComponent(Graphics g) {

```

348


```
122     super.paintComponent(g);
123     long start = alpha.getStartTime();
124     int x1 = 0;
125     int y1 = 150;
126     int x2 = 0;
127     int y2 = 0;
128     for (int i = 1; i < 1000; i++) {
129         x2 = i;
130         y2 = 150 - (int)(100 * alpha.value(start + i * 10));
131         g.drawLine(x1, y1, x2, y2);
132         x1 = x2;
133         y1 = y2;
134     }
135 }
136 }
137 }
```

349

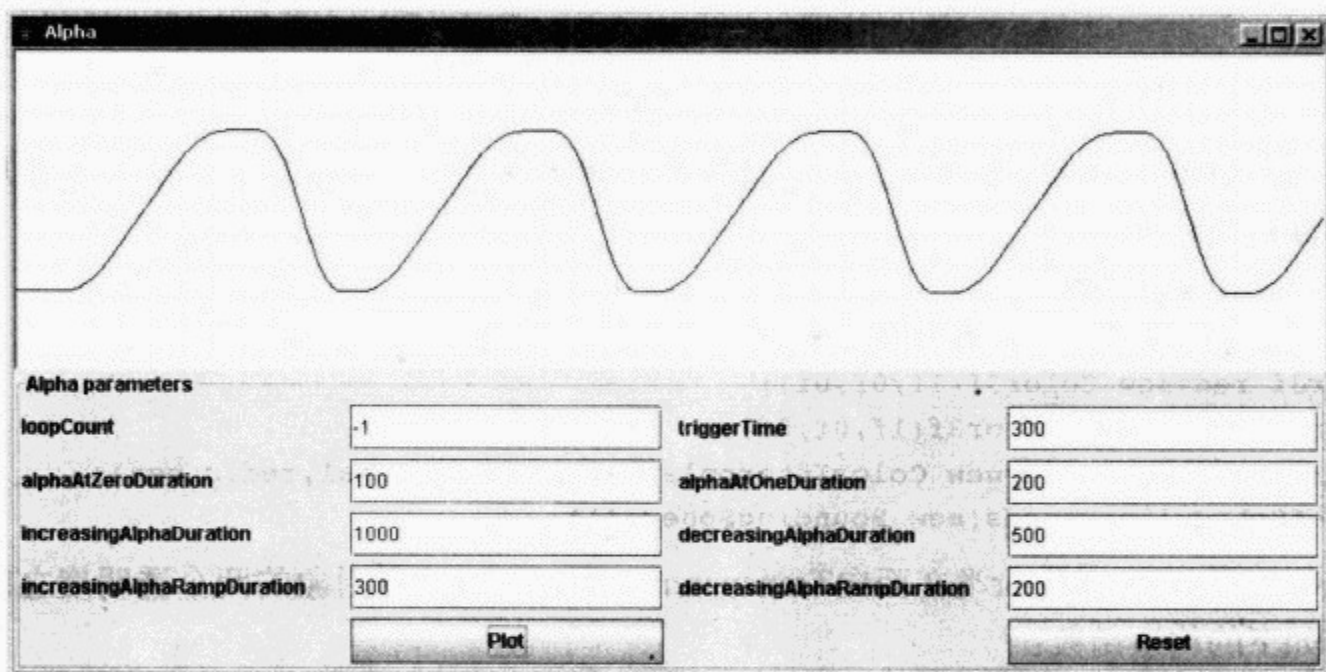


图11-2 绘制Alpha对象的波形

Alpha对象的值由面板输入。alpha对象共有八个参数，它们由框架底层面板的JTextField对象来进行设置。最初，Alpha对象使用了默认的参数值，“Plot”按钮用给定的参数来重画波形，“Reset”按钮将所有的参数重新设置为默认值。

内部Plot类（第114行）继承JPanel类，并在它的paintComponent方法中绘制波形（第121行）。在程序中，每10ms绘制一个点，采用Alpha类对象的value(long atTime)方法来获得alpha值（第130行）。

11.3 插值器

Behavior类包含了一族预定义动画行为的Interpolator子类。Interpolator类为生成动画提供了方便的方法，它先设置了两个或多个属性点，然后在属性点之间进行插值。关于插值器的概念，如图11-3所示，在预先确定的起点和终点两个属性点间，插值器将填充中间值形成动画。插值器的目标可以是颜色、

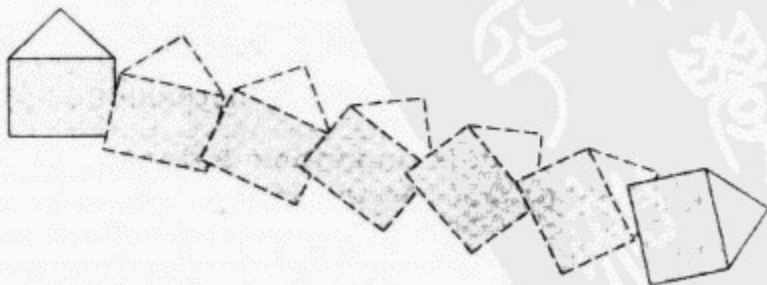


图11-3 一个插值器通过插值生成中间值

变换、切换和其他的属性。

像其他的动画工具一样，插值器也是由时间驱动的。Alpha对象用来标准化动画对象的输入时间，它提供了根据时间值产生激发的一致接口。因此，Java 3D的Interpolator可以由Alpha对象驱动。为了使用插值器创建动画，用户可以先实例化适当的Interpolator类，设置它的控制点并为它提供一个Alpha对象。图11-4显示了Java 3D中Interpolator类的层次结构。

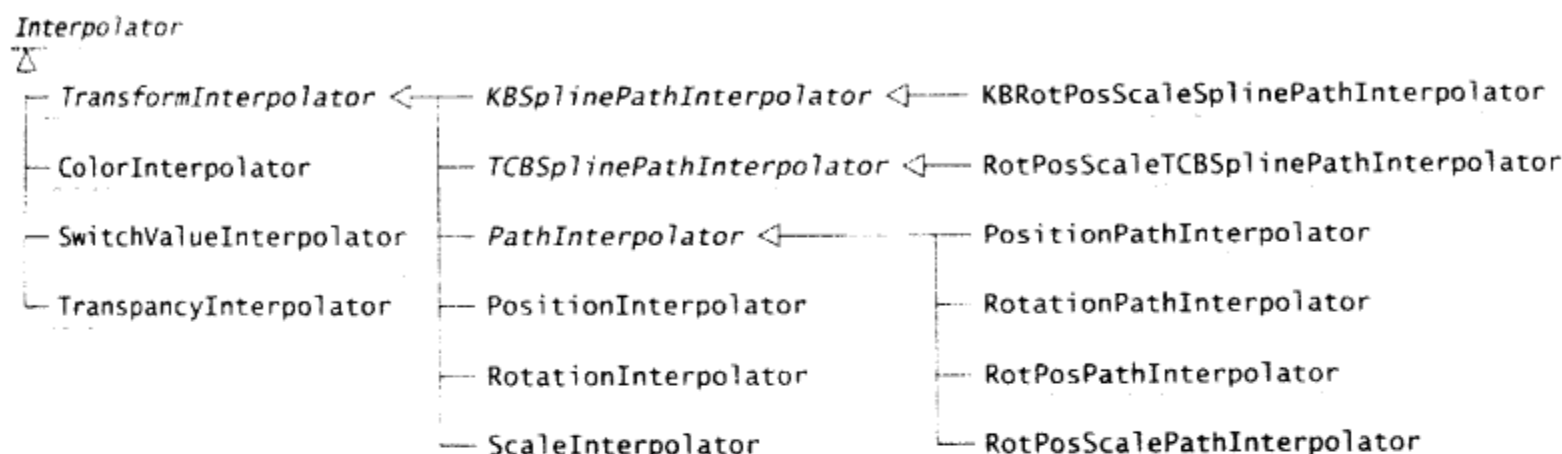


图11-4 插值器

ColorInterpolator类在面向Material对象的两个颜色值之间进行插值。下面给出了从红色到绿色的动画生成代码：

```

Alpha alpha=new Alpha();
Material material=new Material();
Color3f red=new Color3f(1f,0f,0f);
Color3f green=new Color3f(1f,0f,0f);
ColorInterpolator ci=new ColorInterpolator(alpha,material,red,green);
ci.setSchedulingBounds(new BoundingSphere());
  
```

TranspacencyInterpolator类在面向TranspacencyAttributes目标对象的两个透明度之间进行插值。它有如下的构造函数：

```

TranspacencyInterpolator(Alpha a, TranspacencyAttributes target);
TranspacencyInterpolator(Alpha a, TranspacencyAttributes target, float t1,float t2);
  
```

SwitchValueInterpolator类将Switch节点作为它的目标，通过对Switch对象的每个子节点的索引进行插值。

TransformInterpolator类对TransformGroup目标对象进行操作，它通常还包含一个Transform3D对象以表示操作的坐标轴。RotationInterpolator在给定的两个角度之间插值，它的默认值是0~2 π ，在前面的例子中已经多次用到旋转对象。PositionInterpolator沿坐标轴对目标的平移成分进行插值，ScaleInterpolator在缩放的对象间执行插值操作。

有时，对于复杂动画来说，在两个端点间插值是不够的。PathInterpolator类和它的子类是特殊的TransformInterpolator类，它允许指定一序列的控制点或关键帧，插值器在每一对控制点间进行线性插值。与alpha值相关的插值时间由一系列称为节点（knot）的数字进行控制。例如，下面的声明建立了PositionPathInterpolator，它在三个位置之间进行插值：

```

Point3f[] positions={new Point3f(0,0,0),new Point3f(1,1,0),new Point3f(2,0,0)};
float[] knots={0f,0.3f,1f};
Alpha alpha=new Alpha();
TransformGroup target=new TransformGroup();
Transform3D axis=new Transform3D();
  
```



```
PositionPathInterpolator interpolator=
    new PositionPathInterpolator(alpha,target,axis,knots,positions);
```

351

插值路径如图11-5所示。当alpha值在0~0.3之间时，插值器产生从(0,0,0)到(1,1,0)的路径。当alpha值超过0.3时，插值器产生从(1,1,0)到(2,0,0)的路径。

RotationPathInterpolator能够进行一系列的旋转变换的插值，RotPosPathInterpolator能够进行一系列的平移变换和旋转变换的插值，RotPosScaleInterpolator能够进行一系列的平移变换、旋转变换和缩放变换的插值。

PathInterpolator类逐片生成线性动画片段，因此路径在不同分段连接的节点处可能不平滑。为了平滑插值或对插值曲线进行更精确的控制，我们可以采用KBSplinePathInterpolator或TCBSplinePathInterpolator类，它们都有具体的子类，可以进行旋转变换、平移变换和缩放变换的插值。

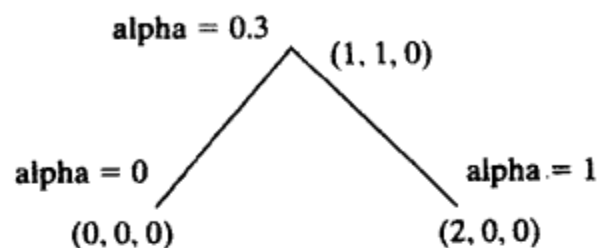


图11-5 一个PositionPathInterpolator

插值器是行为体，因此它需要适当地调整边界集。

程序清单11-2给出了插值的应用。在这个例子中，使用了不同种类的插值器。根据Switch值的不同，可视对象可以是一只海鸥，或者是一个十二面体。在程序生成的界面中，窗口的右边有八个以插值器命名的按钮，点击按钮可激活相应的插值。“Color”按钮进行对象的颜色插值，“Transparency”按钮进行对象的透明度插值，“SwitchValue”按钮进行对象从海鸥到十二面体的转换。“Rotation”、“Position”、“Scale”进行对象的变换操作。“RotPosPath”沿一条路径移动和旋转对象，“RotPosScaleTCBSplinePath”沿一条路径进行平滑插值（如图11-6所示）。

程序清单11-2 TestInterpolator.java

```
1 package chapter11;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.net.*;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.image.*;
11 import com.sun.j3d.utils.behaviors.interpolators.*;
12 import chapter6.Dodecahedron;
13 import chapter10.GullCG;
14 import java.applet.*;
15 import com.sun.j3d.utils.applet.MainFrame;
16 //定义TestInterpolators类，继承自Applet类，用于演示多种插值器
17 public class TestInterpolators extends Applet
18     implements ActionListener {
19     public static void main(String[] args) {
20         new MainFrame(new TestInterpolators(), 800, 600); //创建主窗口并设置大小
21     }
22     //声明8个不同的插值器
23     private ColorInterpolator color = null;
24     private TransparencyInterpolator transparency = null;
25     private SwitchValueInterpolator sw = null;
26     private RotationInterpolator rotator = null;
```

352

```
27 private PositionInterpolator translator = null;
28 private ScaleInterpolator zoom = null;
29 private RotPosPathInterpolator path = null;
30 private TCBSplinePathInterpolator spline = null;
31 private Interpolator current = null;
32 //重写Applet的初始化方法
33 public void init() {
34     setLayout(new BorderLayout()); //设置布局管理器
35     Panel panel = new Panel();
36     panel.setLayout(new GridLayout(8,1));
37     add(panel, BorderLayout.EAST);
38     Button button;
39     button = new Button("Color"); //向面板添加Color按钮,并添加事件侦听器
40     button.addActionListener(this);
41     panel.add(button);
42     button = new Button("Transparency"); //添加Transparency按钮,并添加侦听器
43     button.addActionListener(this);
44     panel.add(button);
45     button = new Button("SwitchValue"); //添加SwitchValue按钮,并添加侦听器
46     button.addActionListener(this);
47     panel.add(button);
48     button = new Button("Rotation"); //添加Rotation按钮,并添加侦听器
49     button.addActionListener(this);
50     panel.add(button);
51     button = new Button("Position"); //添加Position按钮,并添加侦听器
52     button.addActionListener(this);
53     panel.add(button);
54     button = new Button("Scale"); //添加Scale按钮,并添加侦听器
55     button.addActionListener(this);
56     panel.add(button);
57     button = new Button("RotPosPath"); //添加RotPosPath按钮,并添加侦听器
58     button.addActionListener(this);
59     panel.add(button);
60     button = new Button("RotPosScaleTCBSplinePath");
61     button.addActionListener(this);
62     panel.add(button); //添加RotPosScaleTCBSplinePath按钮,并添加侦听器
63     //创建canvas
64     GraphicsConfiguration gc =
65         SimpleUniverse.getPreferredConfiguration();
66     Canvas3D cv = new Canvas3D(gc);
67     add(cv, BorderLayout.CENTER);
68     BranchGroup bg = createSceneGraph();
69     bg.compile();
70     SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
71     su.getViewingPlatform().setNominalViewingTransform();
72     su.addBranchGraph(bg);
73 }
74 //生成BranchGroup的私有方法,用于创建场景图
75 private BranchGroup createSceneGraph() {
76     BranchGroup root = new BranchGroup();
77     TransformGroup tg = new TransformGroup();
78     tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
79     tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
80     root.addChild(tg);
```



```
81 //switch节点
82 Switch swNode = new Switch();
83 swNode.setCapability(Switch.ALLOW_SWITCH_WRITE);
84 tg.addChild(swNode);
85 //外观
86 Appearance ap = new Appearance();
87 Material material = new Material();
88 material.setCapability(Material.ALLOW_COMPONENT_WRITE);
89 material.setColorTarget(Material.AMBIENT);
90 ap.setMaterial(material);
91 TransparencyAttributes transAttr = new TransparencyAttributes(
92 TransparencyAttributes.BLENDED,0.5f);
93 transAttr.setCapability
94 (TransparencyAttributes.ALLOW_VALUE_WRITE);
95 ap.setTransparencyAttributes(transAttr);
96 //gull对象
97 Shape3D shape = new Shape3D(new GullCG(), ap);
98 Transform3D trans = new Transform3D();
99 trans.setScale(0.5);
100 TransformGroup tgScale = new TransformGroup(trans);
101 swNode.addChild(tgScale);
102 tgScale.addChild(shape);
103 //十二面体
104 Dodecahedron dodec = new Dodecahedron();
105 dodec.setAppearance(ap);
106 trans.setScale(0.1);
107 tgScale = new TransformGroup(trans);
108 swNode.addChild(tgScale);
109 tgScale.addChild(dodec);
110 //插值器
111 BoundingSphere bounds =
112 new BoundingSphere(new Point3d(0,0,0),100);
113 Alpha alpha = new Alpha(-1, 6000);
114 alpha.setMode
115 (Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE);
116 alpha.setDecreasingAlphaDuration(6000);
117 //颜色
118 color = new ColorInterpolator
119 (alpha, material, new Color3f(1,0,0), new Color3f(0,0,1));
120 color.setSchedulingBounds(bounds);
121 color.setEnabled(true);
122 root.addChild(color);
123 // 透明度
124 transparency = new TransparencyInterpolator(alpha, transAttr);
125 transparency.setSchedulingBounds(bounds);
126 transparency.setEnabled(false);
127 root.addChild(transparency);
128 // 切换
129 sw = new SwitchValueInterpolator(alpha, swNode);
130 sw.setSchedulingBounds(bounds);
131 transparency.setEnabled(false);
132 root.addChild(sw);
133 // 旋转
134 rotator = new RotationInterpolator(alpha, tg);
```

354

```

135     rotator.setSchedulingBounds(bounds);
136     rotator.setEnable(false);
137     root.addChild(rotator);
138     // 平移
139     translator = new PositionInterpolator(alpha, tg);
140     translator.setSchedulingBounds(bounds);
141     translator.setEnable(false);
142     root.addChild(translator);
143     // 缩放
144     zoom = new ScaleInterpolator(alpha, tg);
145     zoom.setSchedulingBounds(bounds);
146     zoom.setEnable(false);
147     root.addChild(zoom);
148     // 路径
149     Transform3D axis = new Transform3D();
150     float[] knots = {0,0.25f,0.5f,0.75f,1};
151     Quat4f[] rots = new Quat4f[5];
152     Point3f[] ps = new Point3f[5];
153     for (int i = 0; i < 5; i++) {
154         rots[i] = new Quat4f((float)Math.cos(0.5*Math.PI*i),0,0,
155             (float)Math.sin(0.5*Math.PI*i));
156         ps[i] = new Point3f(0.25f*(i-2),
157             (float)Math.sin(0.5*Math.PI*i), 0);
158     }
159     path = new RotPosPathInterpolator
160         (alpha, tg, axis, knots, rots, ps);
161     path.setSchedulingBounds(bounds);
162     path.setEnable(false);
163     root.addChild(path);
164     // 样条
165     TCBKeyFrame[] frames = new TCBKeyFrame[5];
166     for (int i = 0; i < 5; i++) {
167         frames[i] = new TCBKeyFrame(0.25f*i, 0,
168             new Point3f(0.25f*(i-2),(float)Math.sin(0.5*Math.PI*i), 0),
169             new Quat4f((float)Math.cos(0.5*Math.PI*i),0,0,
170                 (float)Math.sin(0.5*Math.PI*i)),
171             new Point3f(1,1,1),0,0,0);
172     }
173     spline = new RotPosScaleTCBSplinePathInterpolator
174         (alpha, tg, axis, frames);
175     spline.setSchedulingBounds(bounds);
176     spline.setEnable(true);
177     root.addChild(spline);
178     current = spline;
179     //设置光照
180     AmbientLight light =
181         new AmbientLight(true, new Color3f(Color.blue)); //添加蓝色环境光源
182     light.setInfluencingBounds(bounds);
183     root.addChild(light);
184     PointLight ptlight = new PointLight(new Color3f(Color.white),
185         new Point3f(0f,0f,2f), new Point3f(1f,0.3f,0f)); //添加白色点光源
186     ptlight.setInfluencingBounds(bounds);
187     root.addChild(ptlight);
188     //设置背景

```



```
189     Background background = new Background(1.0f, 1.0f, 1.0f); //设置背景颜色
190     background.setApplicationBounds(bounds);
191     root.addChild(background);
192     return root;
193 }
194
195 public void actionPerformed(ActionEvent e) { //执行选中按钮的操作
196     String cmd = e.getActionCommand();
197     if ("Rotation".equals(cmd)) {
198         current.setEnable(false);
199         current = rotator;
200         current.setEnable(true);
201     } else if ("Position".equals(cmd)) {
202         current.setEnable(false);
203         current = translator;
204         current.setEnable(true);
205     } else if ("Scale".equals(cmd)) {
206         current.setEnable(false);
207         current = zoom;
208         current.setEnable(true);
209     } else if ("RotPosPath".equals(cmd)) {
210         current.setEnable(false);
211         current = path;
212         current.setEnable(true);
213     } else if ("RotPosScaleTCBSplinePath".equals(cmd)) {
214         current.setEnable(false);
215         current = spline;
216         current.setEnable(true);
217     } else if ("Color".equals(cmd)) {
218         color.setEnable(!color.getEnable());
219     } else if ("Transparency".equals(cmd)) {
220         transparency.setEnable(!transparency.getEnable());
221     } else if ("SwitchValue".equals(cmd)) {
222         sw.setEnable(!sw.getEnable());
223     }
224 }
225 }
```

355

场景图如图11-7所示，两个光源、一个白色背景和一个BoundingSphere对象在这里没有显示。海鸥的形状使用GullCG几何体来表示，Dodecahedron对象通过一些缩放使用Switch节点来连接。对象的外观包含Material和TransparencyAttributes类对象，在Switch的顶部有一个Transformation节点。设置了Transformation、Switch、Material和TransparencyAttributes对象的能力比特，同时允许写这些对象。

一个Alpha对象在创建时开启增长和减少参数，它可以在所有的插值器中使用。程序中创建了八个不同的插值器：ColorInterpolator、TransparencyInterpolator、SwitchValueInterpolator、RotationInterpolator、PositionInterpolator、ScaleInterpolator、RotPosPathInterpolator和RotPosScaleTCBSplinePathInterpolator。ColorInterpolator对Material对象进行操作。TransparencyInterpolator对TransparencyAttributes进行操作，SwitchValueInterpolator对Switch进行操作，另外的五个变换插值器对TransformGroup进行操作。

八个与插值器相对应的按钮在框架的“东”边，所有的按钮都加了动作监听器。前三个按钮能够独立地开启或关闭与它们相关的插值器。另外的五个插值器是互斥的，因为它们是对同

图11-9所示。

程序清单11-3 Pendulum.java

```
1 package chapter11;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.util.*;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import chapter10.ClockBehavior;
11 import java.applet.*;
12 import com.sun.j3d.utils.applet.MainFrame;
13 //定义Pendulum类, 继承自Applet类, 用于演示一个运动的钟摆
14 public class Pendulum extends Applet {
15     public static void main(String[] args) {
16         new MainFrame(new Pendulum(), 480, 640); //创建主窗口并设置窗口大小
17     }
18     //重写Applet的初始化函数
19     public void init() {
20         //创建canvas
21         GraphicsConfiguration gc =
22             SimpleUniverse.getPreferredConfiguration();
23         Canvas3D cv = new Canvas3D(gc);
24         setLayout(new BorderLayout());
25         add(cv, BorderLayout.CENTER);
26         BranchGroup bg = createSceneGraph(); //调用createSceneGraph方法构造场景图
27         bg.compile();
28         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
29         su.getViewingPlatform().setNominalViewingTransform();
30         su.addBranchGraph(bg);
31     }
32     //生成BranchGroup的私有方法, 构造场景图
33     private BranchGroup createSceneGraph() {
34         BranchGroup root = new BranchGroup();
35         //构造钟摆的表面
36         Appearance apFace = new Appearance(); //设置外观
37         Material matFace = new Material();
38         matFace.setAmbientColor(new Color3f(0f, 0f, 0f));
39         matFace.setDiffuseColor(new Color3f(0.15f, 0.15f, 0.25f));
40         apFace.setMaterial(matFace);
41         Cylinder face = new Cylinder(0.6f, 0.01f,
42             Cylinder.GENERATE_NORMALS, 50, 2, apFace); //创建表面圆柱体
43         Transform3D tr = new Transform3D();
44         tr.rotX(Math.PI/2);
45         tr.setTranslation(new Vector3d(0, 0, -0.01));
46         TransformGroup tg = new TransformGroup(tr);
47         tg.addChild(face);
48         root.addChild(tg);
49         //创建时针
50         Appearance ap = new Appearance(); //设置外观
```

358

```

51     ap.setMaterial(new Material());
52     Shape3D shapeHour =
53         new Shape3D(createGeometry(0.4, 0.02, 0.02), ap); //创建时针形体
54     TransformGroup spinHour = new TransformGroup();
55     spinHour.addChild(shapeHour);
56     spinHour.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
57     root.addChild(spinHour);
58     //创建分针
59     Shape3D shapeMin =
60         new Shape3D(createGeometry(0.5, 0.02, 0.02), ap); //创建分针形体
61     TransformGroup spinMin = new TransformGroup();
62     spinMin.addChild(shapeMin);
63     spinMin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
64     root.addChild(spinMin);
65     //创建秒针
66     Shape3D shapeSec =
67         new Shape3D(createGeometry(0.5, 0.01, 0.01), ap); //创建秒针形体
68     TransformGroup spinSec = new TransformGroup();
69     spinSec.addChild(shapeSec);
70     spinSec.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
71     root.addChild(spinSec);
72     //创建行为节点
73     ClockBehavior rotator =
74         new ClockBehavior(spinHour, spinMin, spinSec);
75     BoundingSphere bounds = new BoundingSphere();
76     rotator.setSchedulingBounds(bounds);
77     root.addChild(rotator);
78     //创建钟摆
79     Cylinder rod =
80         new Cylinder(0.01f, 1f, Cylinder.GENERATE_NORMALS, apFace);
81     Transform3D trPend = new Transform3D();
82     trPend.setTranslation(new Vector3d(0, -0.5, -0.01));
83     TransformGroup tgPend = new TransformGroup(trPend);
84     Sphere mass = new Sphere(0.2f, Sphere.GENERATE_NORMALS, 30);
85     Transform3D trMass = new Transform3D();
86     trMass.setScale(new Vector3d(1, 1, 0.2));
87     trMass.setTranslation(new Vector3d(0, -0.5, 0));
88     TransformGroup tgMass = new TransformGroup(trMass);
89     tgMass.addChild(mass);
90     tgPend.addChild(tgMass);
91     TransformGroup tgSwing = new TransformGroup();
92     tgSwing.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
93     tgPend.addChild(rod);
94     tgSwing.addChild(tgPend);
95     root.addChild(tgSwing);
96     Alpha alpha = //创建Alpha对象并设置相应的参数
97         new Alpha(-1, Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE,
98             0, 0, 500, 0, 0, 500, 0, 0);
99     Transform3D trAxis = new Transform3D();
100     trAxis.rotX(Math.PI/2);
101     RotationInterpolator swing = //创建RotationInterpolator对象
102         new RotationInterpolator(alpha, tgSwing, trAxis,
103             (float)(-Math.PI/6), (float)(Math.PI/6));
104     tgPend.addChild(swing);

```



```

105 swing.setSchedulingBounds(bounds);
106 //设置光照
107 AmbientLight light =
108     new AmbientLight(true, new Color3f(Color.blue)); //添加蓝色环境光源
109 light.setInfluencingBounds(bounds);
110 root.addChild(light);
111 PointLight ptlight = new PointLight(new Color3f(Color.white),
112     new Point3f(0.7f, 0.7f, 2f), new Point3f(1f, 0f, 0f)); //添加白色点光源
113 ptlight.setInfluencingBounds(bounds);
114 root.addChild(ptlight);
115 //设置背景
116 Background background = new Background(0.7f, 0.7f, 0.7f); //设置背景颜色
117 background.setApplicationBounds(bounds);
118 root.addChild(background);
119 return root;
120 }
121 //创建GeometryArray的方法
122 GeometryArray createGeometry(double l, double w, double h) {
123     GeometryInfo gi = new GeometryInfo(GeometryInfo.TRIANGLE_ARRAY);
124     Point3d[] pts = new Point3d[4]; //定义点坐标
125     pts[0] = new Point3d(0, 0, h);
126     pts[1] = new Point3d(-w, 0, 0);
127     pts[2] = new Point3d(w, 0, 0);
128     pts[3] = new Point3d(0, l, 0);
129     gi.setCoordinates(pts); //设置点坐标数组
130     int[] indices = {0, 1, 2, 0, 3, 1, 0, 2, 3, 2, 1, 3};
131     gi.setCoordinateIndices(indices); //设置点坐标索引数组
132     NormalGenerator ng = new NormalGenerator();
133     ng.generateNormals(gi); //生成法向量
134     return gi.getGeometryArray();
135 }
136 }

```

359

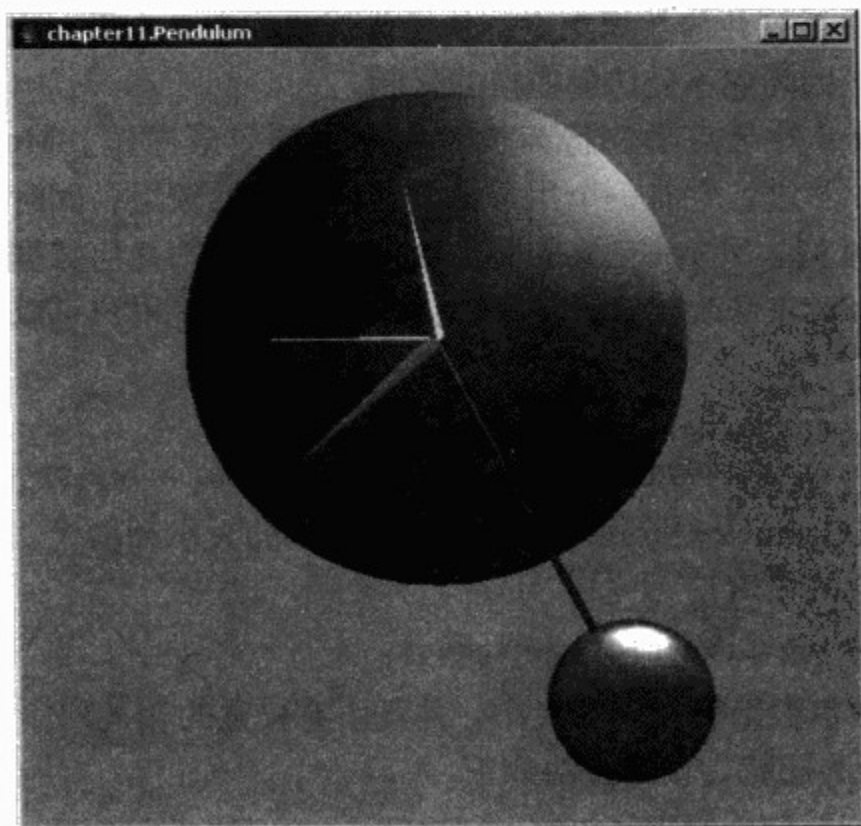


图11-8 钟摆

360

该程序为第10章中定义的模拟时钟增加了一个钟摆，钟摆由一个杆和一个块组成，杆由圆柱体组成，而块由沿z维压缩的球体构成。

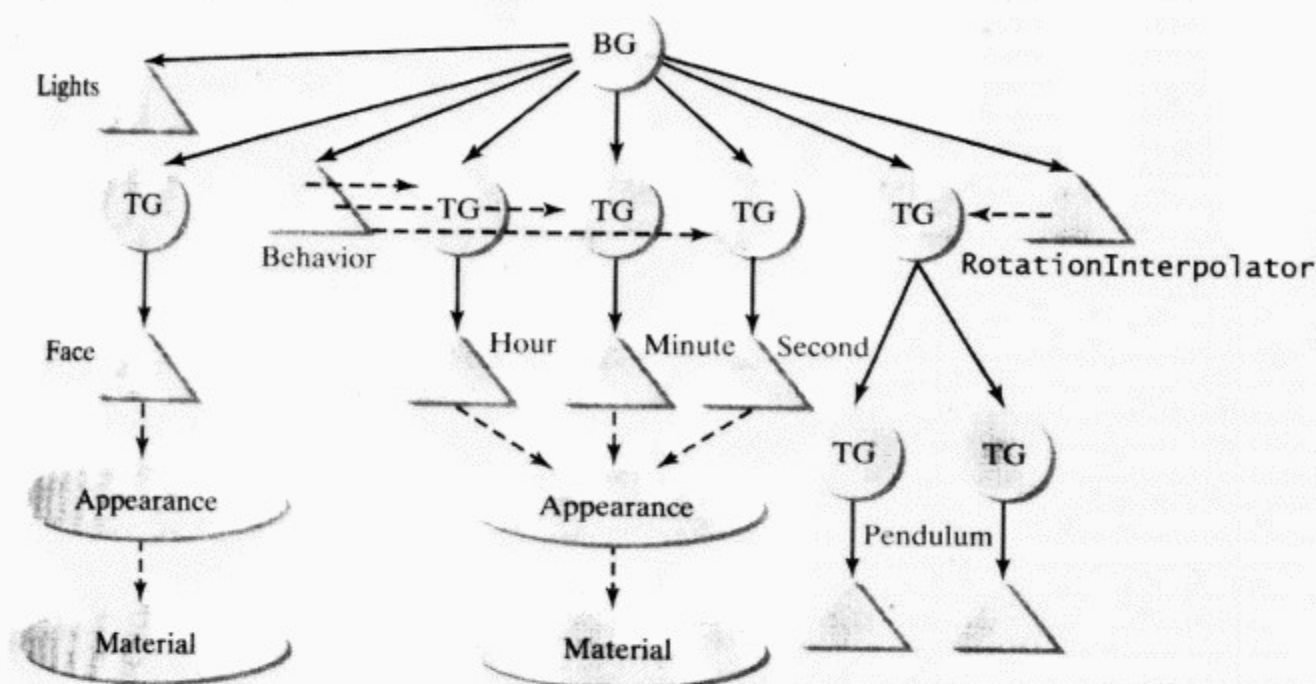


图11-9 增加了钟摆的场景图，钟摆是由插值器驱动的

RotationInterpolator用来摆动钟摆（第102行）。插值器的默认旋转轴是y轴，因为钟摆是在z轴上摆动的，所以要通过Transform3D对象来改变旋转轴。Transform3D对象定义了一个绕x轴旋转 $\pi/2$ 角度的变换，它可以将y轴映射到z轴上。摆动角度限定在 $[-\pi/3, \pi/3]$ ，这可以通过设置RotationInterpolator的两个极限值来实现。

Alpha对象可以设置增长和减少的时间间隔。increasingAlphaDuration和decreasingAlphaDuration都设置为500ms，因此，Alpha波形的周期是1s（第97行）。

11.4 变形

变形（morphing）是一种动画技术，它生成从一个对象平滑地变为另一个对象的视觉效果。变形技术通常应用到一系列几何数据之上。从操作形状的意义上讲，变形比在变换上进行插值操作有更大的灵活性，因为它不受仿射变换的限制。

Java 3D提供Morph叶节点类和行为类对象来支持变形操作。Morph中包含一个外观包（appearance bundle）。不过，它有一个用于提供几何数据的数组，同时，也为对象定义了相应的权重数组。在任何时刻，Morph对象的几何数据由数组中的几何数据的加权和来决定。一种典型例子是由Behavior对象修改权重，从而达到动画的效果。用户可以采用以下的构造函数创建Morph对象：

```
public Morph(GeometryArray[] geometryArrays)
public Morph(GeometryArray[] geometryArrays, Appearance appearance)
```

在Morph中的所有GeometryArray对象都必须有相同的大小和格式。可以调用下面的方法来设置用于合成几何对象的权值：

```
public void setWeights(double[] weights)
```

权重数组和几何数值数组有相同的大小，并且所有权重的和是1.0。为了设置权重，需要设置Morph对象的相应能力比特，设置方法如下：

```
setCapability(Morph.ALLOW_WEIGHTS_WRITE);
```

程序清单11-4给出了一个简单的变形动画，程序清单 11.5给出了描述变形行为的类定义。

这个例子显示了一个形状发生变化的对象，形状变化从一个小圆柱体开始，逐渐平滑地增长对象的高度，然后，向左边弯转而形成一段圆环（如图11-10所示）。这个过程会不断地重复。场景图如图11-11所示。

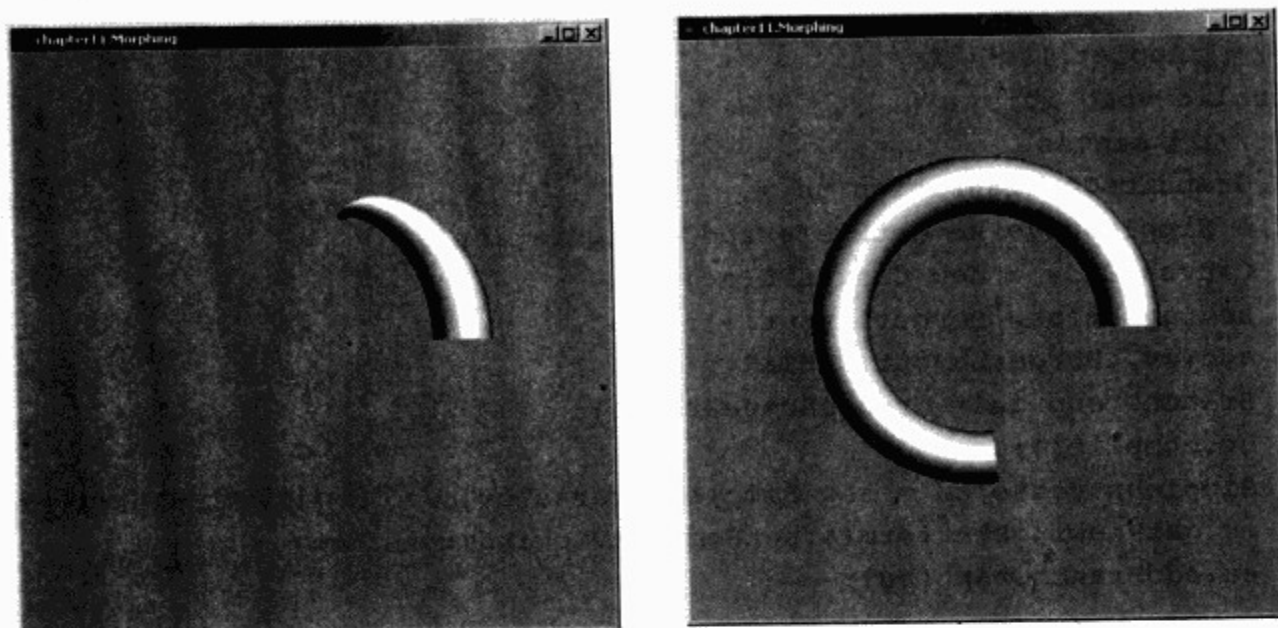


图11-10 变形例子

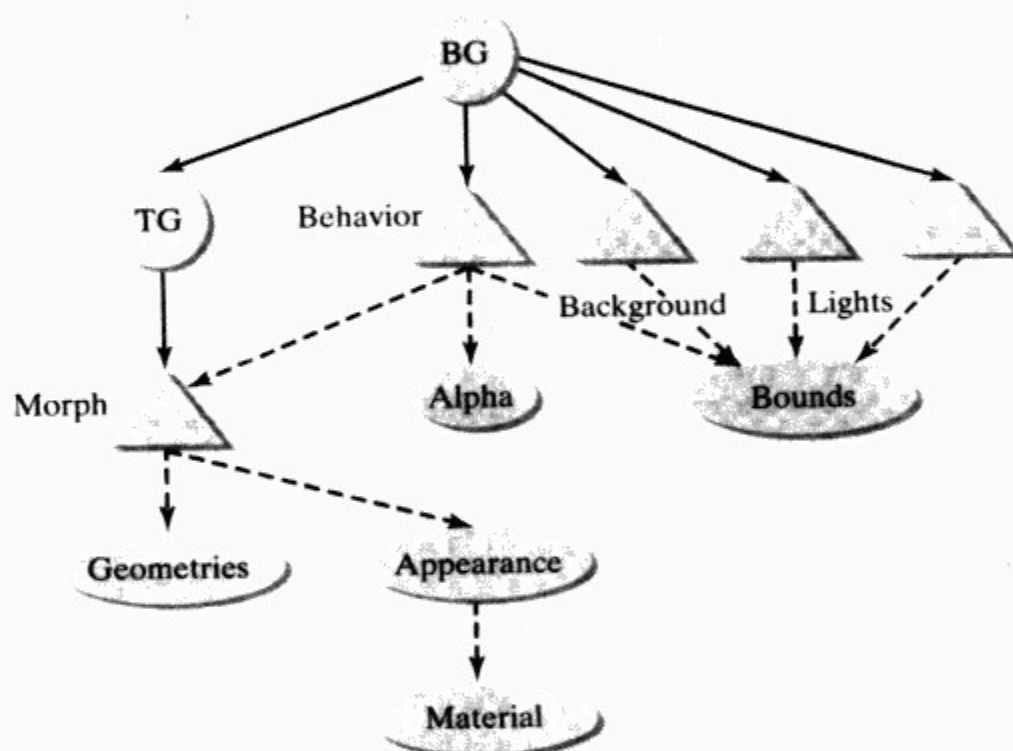


图11-11 变形例子的场景图

程序清单11-4 Morphing.java

```
1 package chapter11;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.util.*;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import java.applet.*;
11 import com.sun.j3d.utils.applet.MainFrame;
```

```
12 //定义Morphing类, 继承自Applet, 演示一个变形动画效果
13 public class Morphing extends Applet {
14     public static void main(String[] args) {
15         new MainFrame(new Morphing(), 480, 480); //创建主窗口并设置大小
16     }
17     //重写Applet初始化函数
18     public void init() {
19         //创建canvas
20         GraphicsConfiguration gc =
21             SimpleUniverse.getPreferredConfiguration();
22         Canvas3D cv = new Canvas3D(gc);
23         setLayout(new BorderLayout());
24         add(cv, BorderLayout.CENTER);
25         BranchGroup bg = createSceneGraph();
26         bg.compile();
27         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
28         su.getViewingPlatform().setNominalViewingTransform();
29         su.addBranchGraph(bg);
30     }
31     //生成BranchGroup对象, 创建场景图
32     private BranchGroup createSceneGraph() {
33         BranchGroup root = new BranchGroup();
34         //创建GeometryArray对象数组
35         GeometryArray[] geoms = new GeometryArray[4];
36         geoms[0] = createGeometry1(0.1);
37         geoms[1] = createGeometry1(0.7);
38         geoms[2] = createGeometry2(0.5);
39         geoms[3] = createGeometry2(0.8);
40         Appearance appear = new Appearance(); //创建外观对象
41         appear.setMaterial(new Material());
42         Morph morph = new Morph(geoms, appear); //创建Morph对象
43         morph.setCapability(Morph.ALLOW_WEIGHTS_READ); //设置Morph对象参数
44         morph.setCapability(Morph.ALLOW_WEIGHTS_WRITE);
45         Transform3D tr = new Transform3D();
46         tr.rotX(Math.PI/2);
47         TransformGroup tg = new TransformGroup(tr);
48         tg.addChild(morph);
49         root.addChild(tg);
50         //创建行为节点
51         Alpha alpha = new Alpha(-1,
52             Alpha.INCREASING_ENABLE|Alpha.DECREASING_ENABLE,
53             0,0, 8000,0,0,8000,0,0); //创建Alpha对象并设置相应的参数
54         MorphingBehavior mb = new MorphingBehavior(morph, alpha);
55         BoundingSphere bounds = new BoundingSphere(); //创建边界对象
56         mb.setSchedulingBounds(bounds);
57         root.addChild(mb);
58         //设置光照
59         AmbientLight light =
60             new AmbientLight(true, new Color3f(Color.blue)); //添加蓝色环境光源
61         light.setInfluencingBounds(bounds);
62         root.addChild(light);
63         PointLight ptlight = new PointLight(new Color3f(Color.white),
64             new Point3f(0.7f,0.7f,2f), new Point3f(1f,0f,0f)); //添加白色点光源
65         ptlight.setInfluencingBounds(bounds);
```



```
66     root.addChild(ptlight);
67     //设置背景
68     Background background = new Background(0.7f, 0.7f, 0.7f); //设置背景颜色
69     background.setApplicationBounds(bounds);
70     root.addChild(background);
71     return root;
72 }
73 //设置变形对象在垂直方向的高度
74 GeometryArray createGeometry1(double h) {
75     double r1 = 0.1;
76     double r2 = 0.5;
77     int m = 20;
78     int n = 40;
79     Point3d[] pts = new Point3d[m];
80     pts[0] = new Point3d(r1+r2, 0, 0);
81     double theta = 2.0 * Math.PI / m;
82     double c = Math.cos(theta);
83     double s = Math.sin(theta);
84     double[] mat = {c, -s, 0, r2*(1-c),
85                     s, c, 0, -r2*s,
86                     0, 0, 1, 0,
87                     0, 0, 0, 1};
88     Transform3D rot1 = new Transform3D(mat);
89     for (int i = 1; i < m; i++) {
90         pts[i] = new Point3d();
91         rot1.transform(pts[i-1], pts[i]);
92     }
93
94     Transform3D rot2 = new Transform3D();
95     rot2.set(new Vector3d(0,0,-h/n));
96     IndexedQuadArray qa =
97         new IndexedQuadArray(m*n, IndexedQuadArray.COORDINATES,
98         4*m*(n-1));
99     int quadIndex = 0;
100    for (int i = 0; i < n; i++) {
101        qa.setCoordinates(i*m, pts);
102        for (int j = 0; j < m; j++) {
103            rot2.transform(pts[j]);
104            int[] quadCoords = {i*m+j, ((i+1)%n)*m+j,
105                                ((i+1)%n)*m+((j+1)%m), i*m+((j+1)%m)};
106            if (i < n-1)
107                qa.setCoordinateIndices(quadIndex, quadCoords); //设置顶点坐标索引
108            quadIndex += 4;
109        }
110    }
111    GeometryInfo gi = new GeometryInfo(qa);
112    NormalGenerator ng = new NormalGenerator();
113    ng.generateNormals(gi); //生成法向量
114    return gi.getGeometryArray();
115 }
116 //设置变形对象在水平方向的长度
117 GeometryArray createGeometry2(double h) {
118     double r1 = 0.1;
119     double r2 = 0.5;
```

```

120     int m = 20;
121     int n = 40;
122     Point3d[] pts = new Point3d[m];
123     pts[0] = new Point3d(r1+r2, 0, 0);
124     double theta = 2.0 * Math.PI / m;
125     double c = Math.cos(theta);
126     double s = Math.sin(theta);
127     double[] mat = {c, -s, 0, r2*(1-c),
128                     s, c, 0, -r2*s,
129                     0, 0, 1, 0,
130                     0, 0, 0, 1};
131     Transform3D rot1 = new Transform3D(mat);
132     for (int i = 1; i < m; i++) {
133         pts[i] = new Point3d();
134         rot1.transform(pts[i-1], pts[i]);
135     }
136
137     Transform3D rot2 = new Transform3D();
138     rot2.rotY(h*2*Math.PI/n);
139     IndexedQuadArray qa =
140         new IndexedQuadArray(m*n, IndexedQuadArray.COORDINATES,
141                               4*m*(n-1));
142     int quadIndex = 0;
143     for (int i = 0; i < n; i++) {
144         qa.setCoordinates(i*m, pts);
145         for (int j = 0; j < m; j++) {
146             rot2.transform(pts[j]);
147             int[] quadCoords = {i*m+j, ((i+1)%n)*m+j,
148                                ((i+1)%n)*m+((j+1)%m),
149                                i*m+((j+1)%m)};
150             if (i < n-1)
151                 qa.setCoordinateIndices(quadIndex, quadCoords);
152             quadIndex += 4;
153         }
154     }
155     GeometryInfo gi = new GeometryInfo(qa);
156     NormalGenerator ng = new NormalGenerator();
157     ng.generateNormals(gi);
158     return gi.getGeometryArray();
159 }
160 }

```

程序清单11-5 MorphingBehavior.java

```

1 package chapter11;
2
3 import javax.media.j3d.*;
4 //定义MorphingBehavior类, 继承自Behavior类, 用于修改Morph节点的权重
5 public class MorphingBehavior extends Behavior {
6     Morph morph; //声明Morph变量
7     Alpha alpha; //声明Alpha变量
8
9     public MorphingBehavior(Morph m, Alpha a) {
10         morph = m;

```



```

11     alpha = a;
12 }
13 //初始化函数
14 public void initialize() {
15     wakeupOn(new WakeupOnElapsedFrames(10));
16 }
17 //修改Morph节点权重的方法
18 public void processStimulus(java.util.Enumeration enumeration) {
19     double[] w = new double[4];
20     double a = alpha.value();
21     w[0] = 0;
22     w[1] = 0;
23     w[2] = 0;
24     w[3] = 0;
25     int index = (int)(a*3);
26     if (index > 2) index = 2;
27     w[index+1] = (a-index/3.0)*3;
28     w[index] = 1.0-w[index+1];
29     morph.setWeights(w);
30     wakeupOn(new WakeupOnElapsedFrames(10));
31 }
32 }

```

程序构造四个GeometryArray对象作为Morph对象的关键帧，这是由createGeometry1和createGeometry2方法创建的。createGeometry1（第74行）创建了具有一定高度的垂直管，createGeometry2(第117行)创建了一段圆环。Morph对象的第一和第二部分几何数据，分别对应于两段具有不同高度的管子，而第三与第四部分的几何内容，则是两段具有不同长度的圆环。

自定义的MorphingBehavior类可以用来修改Morph节点的权重。像插值器一样，权重是基于Alpha值来设置的（第21~29行）。构造函数将Morph对象和Alpha对象作为参数，在initialize和processStimulus方法中唤醒条件设置为10帧。alpha值被分为三个子区间： $[0,1/3]$ 、 $[1/3,2/3]$ 、 $[2/3,1]$ 。在每个子区间上，权重用来插值相邻的几何形状。例如，当alpha值为0.6时，它采用第二个子区间。通过权重的设置来完成第二和第三个几何形状之间的插值，所对应的权重设置如下： $w[0]=0$ ， $w[1]=0.2$ ， $w[2]=0.8$ ， $w[3]=0$ 。

11.5 细节层次

细节层次（level of detail, LOD）是减少绘制复杂形状计算量的一种常用技术，该方法充分注意到关于透视图的一个事实，即在透视图中的离观察者近的可视对象比离观察者远的同一可视对象看起来要大，并且有更多的细节。由此看来，用较少的分辨率和细节来绘制远景对象同时不过多地影响所获得的图像的质量，是可以做到的。

Java 3D提供了LOD抽象类作为Behavior的子类来支持LOD。LOD拥有一个具体的子类DistanceLOD。一个LOD对象在一系列Switch节点上，选择一个Switch子节点作为特定的细节层次。DistanceLOD根据与观察者的距离远近来控制选择，DistanceLOD有如下构造函数：

```

public DistanceLOD()
public DistanceLOD(float[] distances)
public DistanceLOD(float[] distances,Point3f position)

```

其中，distances数组定义了转换到下一层次的临界距离，用于测试到观察者的距离的默认位置是DistanceLOD对象的原点，第三个构造函数允许用户指定一个不同位置。

362
366

程序清单11-6中的程序显示了一个带有纹理映射的地球。用户可以通过鼠标动作来旋转和缩放视图。DistanceLOD对象根据离观察者距离的远近，使用Switch节点来选择具有特定细节层次的球体（如图11-12所示）。

程序清单11-6 TestLOD.java

```
1 package chapter11;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.event.*;
6 import java.awt.image.*;
7 import javax.imageio.*;
8 import com.sun.j3d.utils.geometry.*;
9 import com.sun.j3d.utils.behaviors.vp.*;
10 import com.sun.j3d.utils.universe.*;
11 import javax.media.j3d.*;
12 import javax.vecmath.*;
13 import java.io.*;
14 import java.net.*;
15 import com.sun.j3d.utils.image.*;
16 import java.applet.*;
17 import com.sun.j3d.utils.applet.MainFrame;
18 //定义TestLOD类，继承自Applet类，演示一个带纹理的地球仪
19 public class TestLOD extends Applet {
20     public static void main(String[] args) {
21         new MainFrame(new TestLOD(), 480, 480); //创建主窗口并设置大小
22     }
23
24     BufferedImage[] images = new BufferedImage[3];
25     public void init() { //重写Applet初始化函数
26         //创建canvas
27         GraphicsConfiguration gc =
28             SimpleUniverse.getPreferredConfiguration();
29         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D对象
30         setLayout(new BorderLayout());
31         add(cv, BorderLayout.CENTER);
32         BranchGroup bg = createSceneGraph();
33         bg.compile();
34         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
35         ViewingPlatform viewingPlatform = su.getViewingPlatform();
36         viewingPlatform.setNominalViewingTransform();
37         //创建OrbitBehavior对象，用于缩放和旋转视图
38         OrbitBehavior orbit = new OrbitBehavior(cv,
39             OrbitBehavior.REVERSE_ZOOM |
40             OrbitBehavior.REVERSE_ROTATE |
41             OrbitBehavior.DISABLE_TRANSLATE);
42         BoundingSphere bounds =
43             new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 100.0);
44         orbit.setSchedulingBounds(bounds);
45         viewingPlatform.setViewPlatformBehavior(orbit);
46         su.addBranchGraph(bg);
47     }
48     //生成BranchGroup的方法，创建场景图
```



```
49 public BranchGroup createSceneGraph() {
50     BranchGroup objRoot = new BranchGroup();
51     TransformGroup objTrans = new TransformGroup();
52     objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
53     objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
54     objRoot.addChild(objTrans);
55     //创建Switch对象, 用于保持不同的层次
56     Switch sw = new Switch(0);
57     sw.setCapability(javax.media.j3d.Switch.ALLOW_SWITCH_READ);
58     sw.setCapability(javax.media.j3d.Switch.ALLOW_SWITCH_WRITE);
59     objTrans.addChild(sw);
60     //地球仪的四个层次
61     loadImages();//载入图像
62     Appearance ap = createAppearance(0);//创建外观对象
63     sw.addChild(new Sphere(0.4f,
64         Primitive.GENERATE_TEXTURE_COORDS, 40, ap));
65     ap = createAppearance(1);
66     sw.addChild(new Sphere(0.4f,
67         Primitive.GENERATE_TEXTURE_COORDS, 20, ap));
68     ap = createAppearance(2);
69     sw.addChild(new Sphere(0.4f,
70         Primitive.GENERATE_TEXTURE_COORDS, 10, ap));
71     ap = new Appearance();
72     ap.setColoringAttributes(new ColoringAttributes
73         (0f,0f,0.5f,ColoringAttributes.FATEST));
74     sw.addChild(new Sphere(0.4f, Sphere.GENERATE_NORMALS, 5, ap));
75     //the DistanceLOD behavior
76     float[] distances = new float[3];//创建距离数组distances
77     distances[0] = 5.0f;
78     distances[1] = 10.0f;
79     distances[2] = 25.0f;
80     DistanceLOD lod = new DistanceLOD(distances);//创建LOD对象
81     lod.addSwitch(sw);
82     BoundingSphere bounds = //创建边界对象
83         new BoundingSphere(new Point3d(0.0,0.0,0.0), 10.0);
84     lod.setSchedulingBounds(bounds);
85     objTrans.addChild(lod);
86     //设置背景
87     Background background = new Background(1.0f, 1.0f, 1.0f);
88     background.setApplicationBounds(bounds);
89     objRoot.addChild(background);
90     return objRoot;
91 }
92 //加载图像, 并生成不同分辨率的图像
93 void loadImages() {
94     URL filename =
95         getClass().getClassLoader().getResource("images/earth.jpg");
96     try {
97         images[0] = ImageIO.read(filename);//读取图像
98         AffineTransform xform = //设置仿射变换
99             AffineTransform.getScaleInstance(0.5, 0.5);
100         AffineTransformOp scaleOp = new AffineTransformOp(xform, null);
101         for (int i = 1; i < 3; i++) {
102             images[i] = scaleOp.filter(images[i-1], null);
```



```

103     }
104     } catch (IOException ex) {
105         ex.printStackTrace();
106     }
107 }
108 //创建外观对象, 设置纹理
109 Appearance createAppearance(int i){
110     Appearance appear = new Appearance();
111     ImageComponent2D image =
368     new ImageComponent2D(ImageComponent2D.FORMAT_RGB, images[i]);
112     Texture2D texture =
113     new Texture2D(Texture.BASE_LEVEL, Texture.RGBA,
114         image.getWidth(), image.getHeight());
115     texture.setImage(0, image);
116     texture.setEnabled(true);
117     texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
118     texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
119     appear.setTexture(texture);
120     return appear;
121 }
122 }
123 }

```

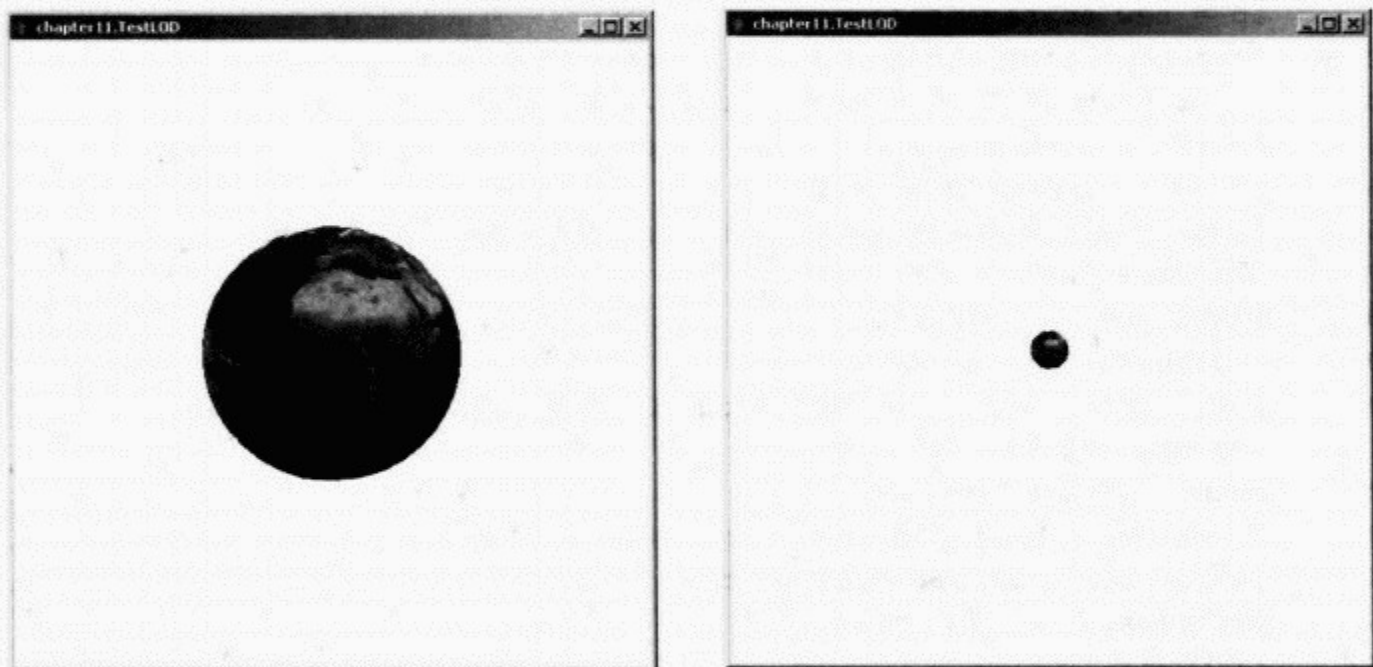


图11-12 LOD的例子

场景图如图11-13所示, 一个有四个子节点的Switch节点, 被创建为DistanceLOD的行为的对象 (第55~81行), 所有的四个子节点都是球体, 但是它们的外观和质地是不同的。开始的三个球体用不同分辨率的图像进行纹理映射, 最后一个球体的外观采用单调的颜色着色。

loadImages方法 (第93行) 为纹理映射准备了三个BufferedImage对象。第一幅图像是从“earth.jpg”文件加载的, 它的大小为512×512。第二幅图像用AffineTransformOp对象, 将第一幅图像缩放为256×256。第三幅图像的获取方式与第二幅图像类似, 只不过是将其缩放为128×128的图像。

对球体的剖分数目也是不同的。第一个球体有40个剖分区域, 它能产生高质量的球体, 而其他三个球体的剖分区域分别为20、10和5。

当球体离观察者近的时候, 用高质量的版本。当它远离观察者时, 就根据LOD行为选择具有较低质量的其他版本的球体。

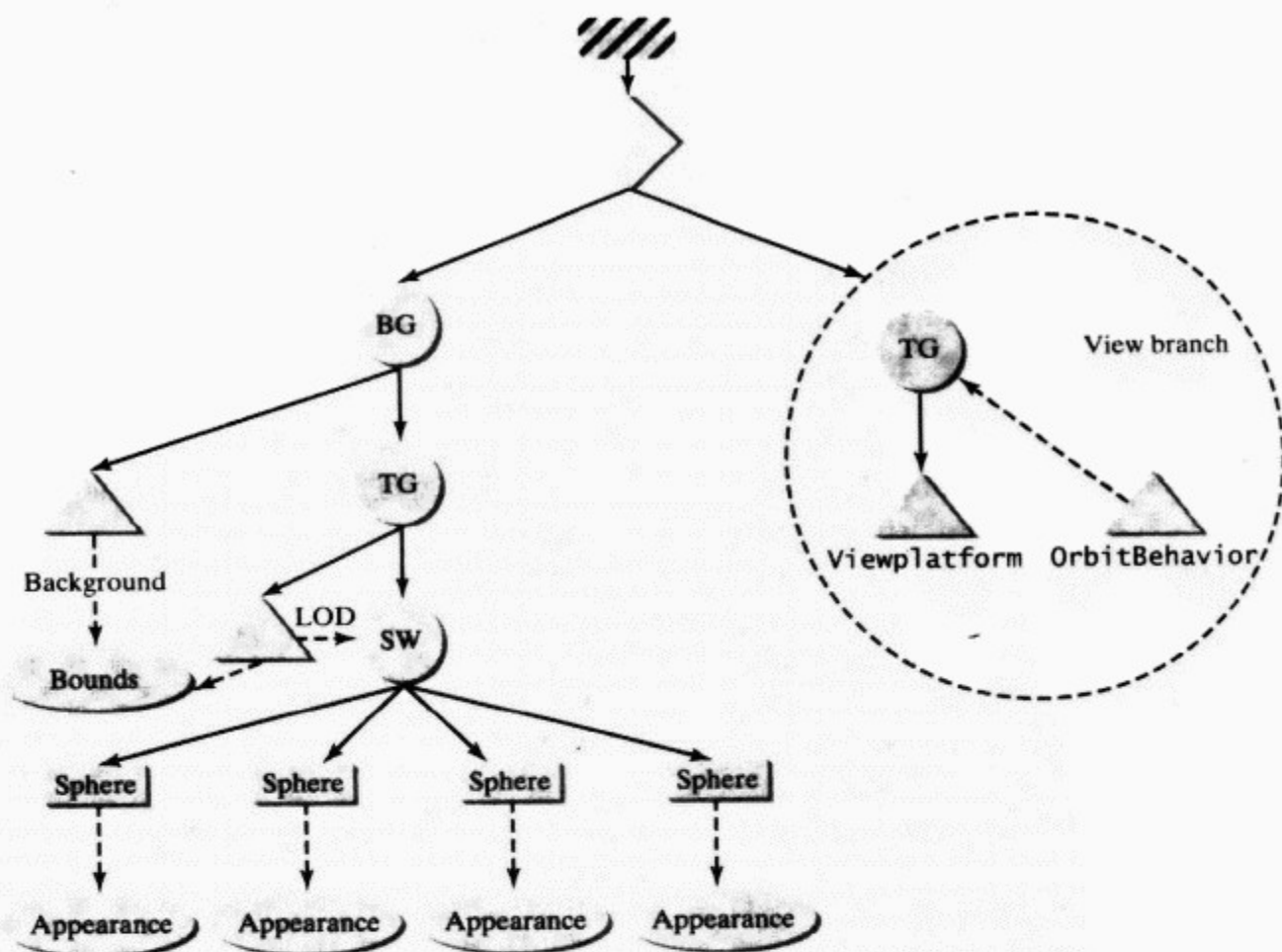


图11-13 LOD例子的场景图

11.6 公告板

Billboard行为在TransformGroup目标对象上实施操作，因此目标的子节点总是面向观察者进行对齐。这种行为机制在目标变换上设置合适的旋转，完成对齐任务，而不用考虑在目标对象顶层的其他变换。

公告板（Billboard）对文本标签之类的可视对象十分有用，它的另一个应用是用2D对象（如图像）模拟一个复杂的3D场景。如果用户看不到图像的“深度”信息，那么公告板行为就能帮助隐藏图像的2D本质。例如，使用3D几何数据来模拟一颗树的代价很高。不过，要是使用2D图像模拟该树，却是相当高效的。因此，对图像使用公告板行为，能达到很好的逼真效果。当然，这种方法只提供树的粗糙“仿制”模型，只有当树大致对称且距离观察者不是很近时，才会有令人满意的效果。

Billboard对象有两种模式：ROTATE_ABOUT_POINT与ROTATE_ABOUT_AXIS。

第一种模式允许使用任意的旋转来形成完全的对齐，第二种模式仅仅允许绕一个轴进行旋转。Billboard类有如下构造函数：

```
public Billboard()
public Billboard (TransformGroup target)
public Billboard (TransformGroup target, int mode, Point3f point)
public Billboard (TransformGroup target, int mode, Vector3f axis)
```

370

程序清单11-7给出的例子，显示了一组分别标记为x、y、z的坐标轴。程序清单11-8给出了一个类，用于描述具有公告板行为的坐标轴。用户可以使用鼠标对视图进行旋转、平移与缩放变换。无论进行何种变换，附加在坐标轴上的3D文本标签，将始终面向观测者（如图11-14所示）。

程序清单11-7 TestBillboard.java

```
1 package chapter11;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.net.URL;
7 import javax.media.j3d.*;
8 import com.sun.j3d.utils.universe.*;
9 import com.sun.j3d.utils.geometry.*;
10 import com.sun.j3d.utils.image.*;
11 import com.sun.j3d.utils.behaviors.mouse.*;
12 import com.sun.j3d.utils.behaviors.vp.*;
13 import java.applet.*;
14 import com.sun.j3d.utils.applet.MainFrame;
15 //定义TestBillboard类,继承自Applet类,显示带有xyz标记的坐标轴
16 public class TestBillboard extends Applet {
17     public static void main(String[] args) {
18         new MainFrame(new TestBillboard(), 480, 480); //创建主窗口并设置大小
19     }
20     //重写Applet的初始化函数
21     public void init() {
22         //创建canvas
23         GraphicsConfiguration gc =
24             SimpleUniverse.getPreferredConfiguration();
25         Canvas3D cv = new Canvas3D(gc);
26         setLayout(new BorderLayout());
27         add(cv, BorderLayout.CENTER);
28         TextArea ta = new TextArea("", 3, 30, TextArea.SCROLLBARS_NONE);
29         ta.setText("Rotation: Drag with left button\n"); //创建并设置文本域
30         ta.append("Translation: Drag with right button\n");
31         ta.append("Zoom: Hold Alt key and drag with left button");
32         ta.setEditable(false); //设置文本域不可编辑
33         add(ta, BorderLayout.SOUTH);
34         BranchGroup root = new BranchGroup();
35         //创建坐标轴
36         Transform3D tr = new Transform3D();
37         tr.setScale(0.5);
38         //tr.setTranslation(new Vector3d(-0.8, -0.7, -0.5));
39         TransformGroup tg = new TransformGroup(tr);
40         root.addChild(tg);
41         AxesBillboard axes = new AxesBillboard(); //创建AxesBillboard对象
42         tg.addChild(axes);
43         BoundingSphere bounds = new BoundingSphere();
44         //设置光照
45         AmbientLight light =
46             new AmbientLight(true, new Color3f(Color.blue)); //添加蓝色环境光源
47         light.setInfluencingBounds(bounds);
48         root.addChild(light);
49         PointLight ptlight = new PointLight(new Color3f(Color.white),
50             new Point3f(0f, 0f, 2f), new Point3f(1f, 0f, 0f)); //添加白色点光源
51         ptlight.setInfluencingBounds(bounds);
52         root.addChild(ptlight);
```

371


```
53 //设置背景
54 Background background = new Background(1.0f, 1.0f, 1.0f); //设置背景颜色
55 background.setApplicationBounds(bounds);
56 root.addChild(background);
57 root.compile();
58 SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
59 su.getViewingPlatform().setNominalViewingTransform();
60 //处理观察平台的运动
61 OrbitBehavior orbit = new OrbitBehavior(cv);
62 orbit.setSchedulingBounds(new BoundingSphere());
63 su.getViewingPlatform().setViewPlatformBehavior(orbit);
64 su.addBranchGraph(root);
65 }
66 }
```

程序清单11-8 AxesBillboard.java

```
1 package chapter11;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 //定义AxesBillboard类, 继承自Group类, 表示带有xyz标记的坐标轴
10 public class AxesBillboard extends Group {
11     public AxesBillboard() {
12         Appearance ap = new Appearance(); //创建外观属性对象
13         ap.setMaterial(new Material()); //设置材质
14         Font3D font = new Font3D(new Font("SanSerif", Font.PLAIN, 1),
15                                 new FontExtrusion()); //创建3D字体
16         Text3D x = new Text3D(font, "x"); //创建3D文本 "x"
17         Shape3D xShape = new Shape3D(x, ap);
18         Text3D y = new Text3D(font, "y"); //创建3D文本 "y"
19         Shape3D yShape = new Shape3D(y, ap);
20         Text3D z = new Text3D(font, "z"); //创建3D文本 "z"
21         Shape3D zShape = new Shape3D(z, ap);
22         //文本的变换节点
23         Transform3D tTr = new Transform3D();
24         tTr.setTranslation(new Vector3d(-0.12, 0.6, -0.04));
25         tTr.setScale(0.5);
26         //箭头的变换节点
27         Transform3D aTr = new Transform3D();
28         aTr.setTranslation(new Vector3d(0, 0.5, 0));
29         // bounds
30         Bounds bounds = new BoundingSphere(new Point3d(0,0,0), 100);
31         //建立x轴形体, 并设置变换节点
32         Cylinder xAxis = new Cylinder(0.05f, 1f);
33         Transform3D xTr = new Transform3D();
34         xTr.setRotation(new AxisAngle4d(0, 0, 1, -Math.PI/2));
35         xTr.setTranslation(new Vector3d(0.5, 0, 0));
36         TransformGroup xTg = new TransformGroup(xTr);
37         xTg.addChild(xAxis);
```

372

```

38     this.addChild(xTg);
39     TransformGroup xTextTg = new TransformGroup(tTr);
40     TransformGroup bbTg = new TransformGroup();
41     bbTg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
42     xTextTg.addChild(bbTg);
43     bbTg.addChild(xShape);
44     Billboard bb = new Billboard(bbTg,
45     Billboard.ROTATE_ABOUT_POINT, new Point3f(0,0,0));
46     bb.setSchedulingBounds(bounds); //创建并设置Billboard对象
47     xTextTg.addChild(bb);
48     xTg.addChild(xTextTg);
49     Cone xArrow = new Cone(0.1f, 0.2f); //创建x轴箭头
50     TransformGroup xArrowTg = new TransformGroup(aTr);
51     xArrowTg.addChild(xArrow);
52     xTg.addChild(xArrowTg);
53     //建立y轴形体, 并设置变换节点
54     Cylinder yAxis = new Cylinder(0.05f, 1f);
55     Transform3D yTr = new Transform3D();
56     yTr.setTranslation(new Vector3d(0, 0.5, 0));
57     TransformGroup yTg = new TransformGroup(yTr);
58     yTg.addChild(yAxis);
59     this.addChild(yTg);
60     TransformGroup yTextTg = new TransformGroup(tTr);
61     bbTg = new TransformGroup();
62     bbTg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
63     yTextTg.addChild(bbTg);
64     bbTg.addChild(yShape);
65     bb = new Billboard(bbTg,
66     Billboard.ROTATE_ABOUT_POINT, new Point3f(0,0,0));
67     bb.setSchedulingBounds(bounds); //创建并设置Billboard对象
68     yTextTg.addChild(bb);
69     yTg.addChild(yTextTg);
70     Cone yArrow = new Cone(0.1f, 0.2f); //创建y轴箭头
71     TransformGroup yArrowTg = new TransformGroup(aTr);
72     yArrowTg.addChild(yArrow);
73     yTg.addChild(yArrowTg);
74     //建立z轴形体, 并设置变换节点
75     Cylinder zAxis = new Cylinder(0.05f, 1f);
76     Transform3D zTr = new Transform3D();
77     zTr.setRotation(new AxisAngle4d(1, 0, 0, Math.PI/2));
78     zTr.setTranslation(new Vector3d(0, 0, 0.5));
79     TransformGroup zTg = new TransformGroup(zTr);
80     zTg.addChild(zAxis);
81     this.addChild(zTg);
82     TransformGroup zTextTg = new TransformGroup(tTr);
83     bbTg = new TransformGroup();
84     bbTg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
85     zTextTg.addChild(bbTg);
86     bbTg.addChild(zShape);
87     bb = new Billboard(bbTg,
88     Billboard.ROTATE_ABOUT_POINT, new Point3f(0,0,0));
89     bb.setSchedulingBounds(bounds); //创建并设置Billboard对象
90     zTextTg.addChild(bb);
91     zTg.addChild(zTextTg);

```



```

92     Cone zArrow = new Cone(0.1f, 0.2f); //创建z轴箭头
93     TransformGroup zArrowTg = new TransformGroup(aTr);
94     zArrowTg.addChild(zArrow);
95     zTg.addChild(zArrowTg);
96 }
97 }

```

373

所显示的对象是一组坐标轴，这与在第7章里定义的Axes类很相像。不同之处在于，这里在场景图中增加了Billboard行为节点（第44、65、87行）和TransformGroup节点，如图11-15所示。Billboard节点是使用ROTATE_ABOUT_POINT模式创建的。

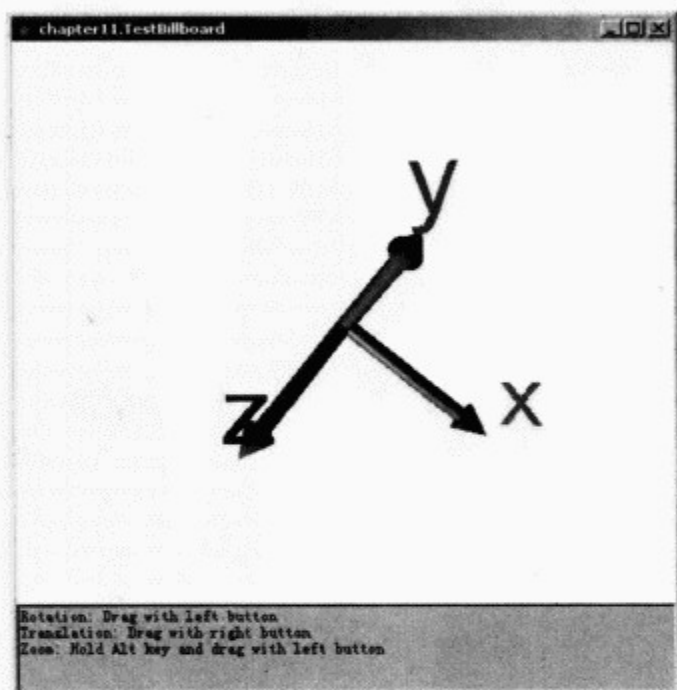


图11-14 公告板行为的例子

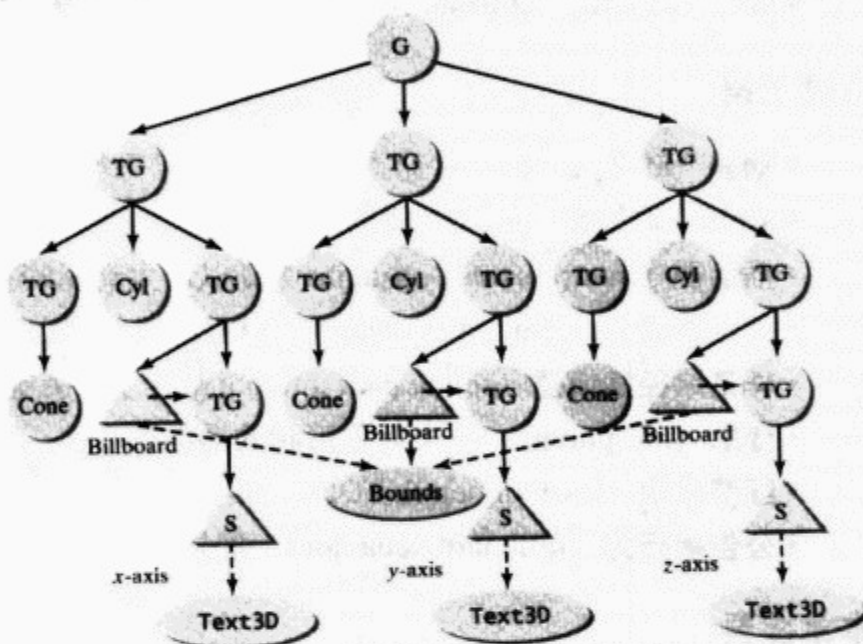


图11-15 AxesBillboard的场景图

374

程序显示了带有光照的AxesBillboard类的一个实例。视图有一个OrbitBehavior对象，允许使用鼠标进行旋转、平移和缩放变换。在各种变换中，“x”、“y”与“z”标签将与坐标轴一起移动。不过，由于公告板行为的特性，文本标签将始终面向观测者。

需要注意到，尽管Billboard对象只需要改变变换的旋转分量，但是它会对平移和缩放分量进行清除。因此，为Billboard创建另外的TransformGroup节点是十分必要的。

主要的类和方法

- javax.media.j3d.Alpha 封装了alpha函数的类。
- javax.media.j3d.Interpolator 插值器行为的基类。
- javax.media.j3d.ColorInterpolator 在Material对象上对颜色进行插值操作。
- javax.media.j3d.TransparencyInterpolator 在TransparencyAttributes对象上进行插值操作。
- javax.media.j3d.SwitchValueInterpolator 在Switch对象上进行插值操作。
- javax.media.j3d.TransformInterpolator 变换插值器的基类。
- javax.media.j3d.RotationInterpolator 旋转插值器。
- javax.media.j3d.PositionInterpolator 平移插值器。
- javax.media.j3d.ScaleInterpolator 缩放插值器。
- javax.media.j3d.PathInterpolator 由帧序列定义的变换插值器的基类。
- javax.media.j3d.RotationPathInterpolator 旋转变换的路径插值器。
- javax.media.j3d.PositionPathInterpolator 平移变换的路径插值器。

- `javax.media.j3d.RotPosPathInterpolator` 旋转和平移变换的路径插值器。
- `javax.media.j3d.RotPosScalePathInterpolator` 旋转、平移和缩放变换的路径插值器。
- `com.sun.j3d.utils.behaviors.interpolator.RotPosScaleTCBSplinePathInterpolator` 旋转、平移和缩放变换的样条路径插值器。
- `com.sun.j3d.utils.behaviors.interpolators.KBRotPosScaleSplinePathInterpolator` 旋转、平移和缩放变换的样条路径插值器。
- `javax.media.j3d.Morph` 绘制几何形状数组的叶节点类。
- `javax.media.j3d.LOD` LOD行为类。
- `javax.media.j3d.DistanceLOD` 操作于基于距离的Switch节点的LOC类。
- `javax.media.j3d.Billboard` 调整对象使之面向观测者的行为类。

关键术语

- 动画 (animation) 对动态场景中的一系列帧进行绘制的过程。
- alpha 取值在0.0~1.0之间的时间函数，它用于驱动插值器。
- 插值器 (interpolator) 由alpha驱动以产生动画的行为。
- 路径插值器 (path interpolator) 由关键帧序列定义的变换插值器。
- 样条路径插值器 (spline path interpolator) 由关键帧序列定义的、沿平滑路径上的变换插值器。
- 变形 (morphing) 一个对象形状的动态改变。
- 细节层次 (level of detail, LOD) 根据视图的情况改变绘制细节层次的技术。
- 公告板行为 (billboard behavior) 能够自动地使一个对象朝向观测者的行为。

本章提要

- 本章介绍了常用的动画技术和Java 3D的行为动画支持功能。
- 插值器是一个方便的动画工具，它由Alpha对象驱动，在指定的控制点间或关键帧间插值。Java 3D中包括了大量的插值器集合，能对不同的属性（从颜色到样条曲线路径）进行插值。
- Morph节点涉及到一系列几何数据，通过对几何数据进行加权组合，来生成新的对象。一般地，通过行为对象能够改变权重的大小，从而生成从一个对象到另一个对象间的特殊变形动画。
- LOD行为控制Switch节点，选择具有不同细节层次的对象。LOD提供的工具能自动减少不必要绘制的细节的数量，这种方法有助于在不显著降低绘制质量的情况下，提高绘制效率。
- Billboard行为能自动地旋转对象，使它始终面向观测者，它在设置文本标签和由2D图像模拟3D对象时很有效。

复习题

11.1 用以下给定的参数画出Alpha对象的波形：

```

LoopCount=5
TriggerTime=0
PhaseDelayDuration=100
AlphaAtZeroDuration=0
AlphaAtOneDuration=200
IncreasingAlphaDuration=0
DecreasingAlphaDuration=200
IncreasingAlphaRampDuration=0
DecreasingAlphaRampDuration=0

```

11.2 用以下给定的参数画出Alpha对象的波形：

```

LoopCount=-1

```



```
TriggerTime=0
PhaseDelayDuration=0
AlphaAtZeroDuration=200
AlphaAtOneDuration=200
IncreasingAlphaDuration=400
DecreasingAlphaDuration=200
IncreasingAlphaRampDuration=100
DecreasingAlphaRampDuration=0
```

11.3 找出图11-16中Alpha对象波形的参数。



图11-16 Alpha波形

11.4 程序清单11-3中使用了RotationInterpolator来摆动钟摆，还有其他的插值器能够完成这项任务吗？

376

11.5 除了基于距离的LOD，你能想到恰当使用LOD技术的其他例子吗？

11.6 讨论Billboard中ROTATE_ABOUT_AXIS模式的局限性，以及在什么情况下能使用该模型。

编程练习

11.1 在程序清单11-1中，在面板上添加Alpha的另两个参数mode和phaseDelayDuration的按钮。

11.2 编写Java 3D程序，实现一个四面体，并不断地从左到右移动它。

11.3 编写Java 3D程序，实现四面体沿着三角形的边不断运动。三角形的顶点分别为 $(-0.5, 0, 0)$ 、 $(0.5, -0.5, 0)$ 、 $(0, 0, 0.5)$ 。

11.4 在程序清单11-2中，添加如下的变换插值器：

```
RotationPathInterpolator
PositionPathInterpolator
RotPosScalePathInterpolator
KBRotPosScaleSplinePathInterpolator
```

11.5 编写程序，用Morph对象和Behavior对象来实现物体从圆锥体到圆柱体间的变形。

11.6 编写程序，用DistanceLOD对象建立三个细节层次，来显示3D字符串“Java”。第一层在外观中使用Material对象来设置光照，第二层用单调颜色进行着色，第三层用填充的矩形来表示文本。

11.7 编写Java 3D程序，用Billboard行为实现显示纹理影射的矩形。使用OrbitBehavior对象，允许通过鼠标操作来操纵视图。

377

第12章 其他3D主题

学习目标

- 定义与实现3D曲线。
- 定义与实现3D曲面。
- 在Java 3D场景图中使用声音。
- 创建简单的阴影。
- 理解动态几何变化。
- 通过离屏绘制捕捉绘制图像。
- 使用3D纹理映射。
- 理解与实现纹理合成。

378
379

12.1 引言

本章将介绍几种与3D图形有关的高级技术。

3D曲线和曲面是3D几何的重要组成部分，当前版本的Java 3D并没有提供对3D曲线和3D曲面的直接支持。在本章中，我们将介绍一些构造3D曲线和曲面的例子。本章还将介绍用于Bézier曲线的求值与细分的deCasteljau算法。

许多现实应用需要将声音和图形组合起来使用。Java 3D支持在场景图中直接嵌入声音，使图形对象和声音的组合变得轻而易举。

在计算机图形学中，阴影是一种非常复杂的对象。Java 3D所使用的局部光照模型无法自动产生阴影，本章将介绍一种用多边形对象产生人工阴影的简单方法。

本书前面所介绍的动画和交互应用例子，通常都不涉及对几何对象的动态修改，但活动场景图的几何数据还是可以改变的，尽管这需要一种特殊的处理过程。首先，GeometryArray类对象必须以BY_REFERENCE模式创建，并且几何数据的改动只能通过一个实现了GeometryUpdater接口的类来进行。我们将通过一个动态阴影的例子来介绍这一过程。

在很多应用中，我们需要从Java 3D场景的Canvas3D对象获取场景的绘制图像。本章将介绍Canvas3D类的离屏绘制功能，我们可以使用该功能来捕捉一帧画面，并将其保存到一个BufferedImage类对象中。

3D纹理映射把3D立体图像映射到图形对象上，它能够创造出2D纹理映射难以达到的视觉真实感。Java 3D为3D纹理映射提供了直接的支持，与此相关的一个话题是合成纹理。Perlin噪音函数是一个可以表现出一定光滑性的随机函数，我们可以用它来生成非常接近真实情况的大理石、木料或金属等自然纹理。

12.2 3D曲线

3D空间中的Bézier曲线（Bézier curves）和B样条曲线的定义，与2D空间中的情况相同。控制点（control points）为 p_0, p_1, \dots, p_n 的一条Bézier曲线的参数方程为

$$s(t) = \sum_{i=0}^n p_i B_{n,i}(t)$$

其中 $B_{n,i}(t)$ 称为Bernstein多项式 (Bernstein polynomial) 或Bernstein基 (Bernstein basis)

$$B_{n,i}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

特别的, 三次Bézier曲线的度数 $n=3$, 由四个控制点定义

$$s(t) = (1-t)^3 p_0 + 3t(1-t)^2 p_1 + 3t^2(1-t)p_2 + t^3 p_3$$

deCasteljau算法 (deCasteljau algorithm) 以线性插值的方式从控制点出发, 计算曲线上任意一点的坐标, 该算法构造了一个类似于Pascal三角的三角阵。计算三次Bézier曲线上的点, 使用的是如下的三角阵:

$$\begin{array}{c} p_0^0, p_1^0, p_2^0, p_3^0 \\ p_0^1, p_1^1, p_2^1 \\ p_0^2, p_1^2 \\ p_0^3 \end{array}$$

第一行中包含的是Bézier曲线的原始控制点, 随后的每一行都是用以下公式计算的:

$$p_i^k = (1-t)p_i^{k-1} + tp_{i+1}^{k-1}$$

380

三角阵的最后一项, 就是该Bézier曲线在 t 处的点

$$s(t) = p_0^3$$

图12-1图示出了整个计算过程, deCasteljau算法涉及的所有计算都只是简单的线性插值。

每条Bézier曲线都可以分割成两条子Bézier曲线。deCasteljau算法除了能计算曲线上的任意点, 同时还给出了一种用于细分 (subdivision) Bézier曲线的方法, 如图12-1所示的三次曲线被细分成两条曲线, 它们的控制点分别为 $p_0^0, p_0^1, p_0^2, p_0^3$ 和 $p_0^3, p_1^2, p_1^1, p_1^0$ 。

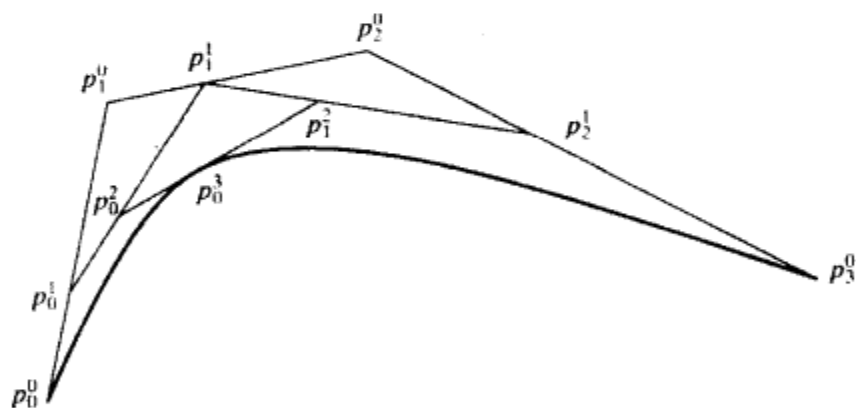


图12-1 deCasteljau算法

当前的Java 3D (1.3版) 并未包含对曲线和曲面的直接支持, 我们将用Java 3D提

供的直线对象和多边形数组对象来实现我们自己版本的Bézier曲线和曲面。其他类型的样条曲线与曲面则可以用一系列的Bézier曲线或曲面来实现。

程序清单12-1以递归调用的方式绘制三次Bézier曲线, 程序清单12-2是一个显示Bézier曲线的测试程序。我们实现了BezierCurve类, 它是LineStripArray类的子类, 测试程序则显示了此曲线类的一个实例对象 (如图12-2所示)。

程序清单12-1 BezierCurve.java

```
1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;
```

```

5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 //定义BezierCurve类,继承自LineStripArray类,表示Bezier曲线
10 public class BezierCurve extends LineStripArray {
11     static int level = 4; //固定的最大递归调用层数
12     static int[] vCnts = {(1<<level)+1};
13     int index = 0; //当前将设置的直线段端点的序号
14     // BezierCurve构造函数,参数为4个控制点
15     public BezierCurve(Point3d p0,Point3d p1,Point3d p2,Point3d p3) {
16         super(vCnts[0], GeometryArray.COORDINATES, vCnts); //调用父类构造函数
17         setCoordinate(index, p0); //设置第一个直线段的第一个端点
18         index++;
19         subdivide(0,p0,p1,p2,p3); //递归调用subdivide方法
20     }
21     //对以参数中四个点为控制点的曲线段进行分段
22     void subdivide(int lev, Point3d p0, Point3d p1,
23     Point3d p2, Point3d p3) {
24         if (lev >= level){ //如果已经达到最大递归调用层数
25             setCoordinate(index, p3); //设置LineStripArray对象中的直线段端点
26             index++;
27         }
28         else {
29             Point3d p10 = new Point3d();
30             p10.add(p0,p1);
31             p10.scale(0.5); //点p10设置为点p0和点p1的插值中点
32             Point3d p11 = new Point3d();
33             p11.add(p1,p2);
34             p11.scale(0.5); //点p11设置为点p1和点p2的插值中点
35             Point3d p12 = new Point3d();
36             p12.add(p2,p3);
37             p12.scale(0.5); //点p12设置为点p2和点p3的插值中点
38             Point3d p20 = new Point3d();
39             p20.add(p10,p11);
40             p20.scale(0.5); //点p20设置为点p10和点p11的插值中点
41             Point3d p21 = new Point3d();
42             p21.add(p11,p12);
43             p21.scale(0.5); //点p21设置为点p11和点p12的插值中点
44             Point3d p30 = new Point3d();
45             p30.add(p20,p21);
46             p30.scale(0.5); //点p30设置为点p20和点p21的插值中点
47             subdivide(lev+1,p0,p10,p20,p30); //以新的控制点递归调用subdivide方法
48             subdivide(lev+1,p30,p21,p12,p3); //以新的控制点递归调用subdivide方法
49         }
50     }
51 }

```

程序清单12-2 TestBezierCurve.java

```

1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;

```



```
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义TestBezierCurve类,继承自Applet类,演示使用BezierCurve类
12 public class TestBezierCurve extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new TestBezierCurve(), 640, 480); //创建主窗口并设置大小
15     }
16     //重写Applet初始化方法
17     public void init() {
18         //创建Canvas3D画布对象
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
21         Canvas3D cv = new Canvas3D(gc);
22         setLayout(new BorderLayout()); //设置布局管理器
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph(); //创建场景图分支
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
27         su.getViewingPlatform().setNominalViewingTransform();
28         su.addBranchGraph(bg);
29     }
30     //生成BranchGroup的私有方法,创建场景图分支
31     private BranchGroup createSceneGraph() {
32         BranchGroup root = new BranchGroup(); //创建分支根节点
33         TransformGroup spin = new TransformGroup();
34         spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
35         root.addChild(spin);
36         //创建形体
37         Point3d p0 = new Point3d(-1,0,0.5);
38         Point3d p1 = new Point3d(-0.2,0.6,-0.2);
39         Point3d p2 = new Point3d(0.3,-0.8,0.3);
40         Point3d p3 = new Point3d(0.9,0.1,0.6);
41         Appearance ap = new Appearance(); //创建外观对象
42         ap.setColoringAttributes(new ColoringAttributes(0f, 0f, 0f,
43             ColoringAttributes.FASTEST));
44         Shape3D shape = new Shape3D(new
45             BezierCurve(p0,p1,p2,p3), ap); //创建Bezier曲线
46         spin.addChild(shape);
47         //设置旋转
48         Alpha alpha = new Alpha(-1, 10000);
49         RotationInterpolator rotator =
50             new RotationInterpolator(alpha, spin);
51         BoundingSphere bounds = new BoundingSphere(); //球体作用范围边界对象
52         rotator.setSchedulingBounds(bounds);
53         spin.addChild(rotator);
54         //设置背景
55         Background background = new Background(1f, 1f, 1f); //设置背景颜色
56         background.setApplicationBounds(bounds);
57         root.addChild(background);
```



```

58     return root;
59 }
60 }

```

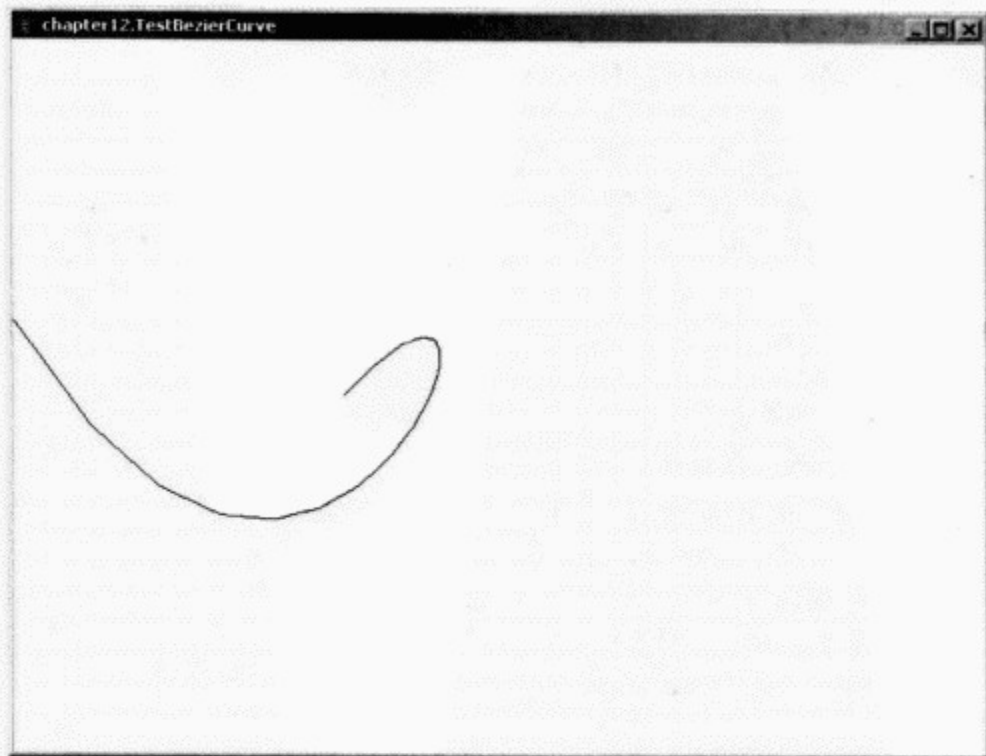


图12-2 一条三次Bézier曲线

我们用BezierCurve类来实现Bézier曲线，前者是LineStripArray类的一个子类。该程序递归地用一种基于deCasteljau算法的细分技术，把曲线细分成小片段，最终形成的Bézier曲线小段，则用直线段来近似。为了简化实现，该程序使用固定层数的递归，且没有对各分段进行任何平滑性测试。

如果递归的深度为 $level$ ，则最终的直线将被分为 2^{level} 个线段，而相应的LineStripArray对象的顶点数为 $2^{level}+1$ 。BezierCurve类的构造函数首先把第一个点加入此数组中，然后调用subdivide方法（第22行）开始对曲线进行细分。

subdivide方法是一个递归方法，它在 $t = 0.5$ 处把一条Bézier曲线分成两段，然后针对这两段曲线分别再递归调用方法自身。当递归调用层数达到预定的最大值时（在本例中设为4），将停止进一步的细分，并把该段的另一个端点加入数组。

测试程序TestBezierCurve将显示一个Bézier曲线实例，并用旋转插值器使Bézier曲线旋转起来。

12.3 曲面

12.3.1 Bézier曲面

把一条Bézier曲线沿另一条Bézier曲线用一种称为张量积（tensor product）的方法进行延伸，就得到了一个Bézier曲面。Bézier曲面的参数方程为：

$$s(u, v) = \sum_{i=0}^m \sum_{j=0}^n p_{i,j} B_{m,i}(u) B_{n,j}(v)$$

使用最为广泛的一种Bézier曲面是 $m = n = 3$ 的三次Bézier曲面，这样的曲面需要16个控制点来定义。

Bézier曲面上任意一点 $S(u, v)$ 的求值（evaluation），也可以用deCasteljau算法来计算。每个Bézier曲面都可以看做是一族Bézier曲线。对一个确定的 u 来说，曲线 $s(v) = S(u, v)$ 就是一条

Bézier曲线，其控制点为：

$$p_j = \sum_{i=0}^3 p_{i,j} B_{3,i}(u), \quad j=0,1,2,3$$

这四个控制点本身是另一条Bézier曲线上的点。因此，以 $t = u$ 四次调用deCasteljau算法，就可以计算得到这些点。得到此Bézier曲线上的四个控制点之后，就可以用 $t = v$ 再次调用deCasteljau算法来计算点 $s(v) = S(u,v)$ 。

程序清单12-3实现了一个双三次Bézier曲面，程序清单12-4中的测试程序，则显示了一个随机生成的Bézier曲面（如图12-3所示）。

程序清单12-3 BezierSurface.java

```

1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.geometry.*;
8 //定义BezierSurface类，继承自Shape3D类，定义Bezier曲面
9 public class BezierSurface extends Shape3D{
10     public BezierSurface(Point3d[][] ctrlPts) { //根据控制点参数构造Bezier曲面
11         int m = 17;
12         int n = 17;
13         Point3d[] pts = new Point3d[m*n]; //用于存放Bezier曲面上的m*n个点
14         int idx = 0;
15         Point3d[] p = new Point3d[4];
16         double du = 1.0/(m-1);
17         double dv = 1.0/(n-1);
18         double u = 0;
19         double v = 0;
20         for (int i = 0; i < m; i++) {
21             for (int k = 0; k < 4; k++) { //计算四条Bezier曲线在u处的点
22                 p[k] = deCasteljau(u, ctrlPts[k]);
23             }
24             v = 0;
25             for (int j = 0; j < n; j++) { //以p为控制点定义一条新的Bezier曲线，
26                 pts[idx++] = deCasteljau(v, p); //并求此曲线在v处的点
27                 v += dv;
28             }
29             u += du;
30         }
31
32         int[] coords = new int[2*n*(m-1)]; //顶点坐标索引数组
33         idx = 0;
34         for (int i = 1; i < m; i++) {
35             for (int j = 0; j < n; j++) {
36                 coords[idx++] = i*n + j;
37                 coords[idx++] = (i-1)*n + j;
38             }
39         }
40
41         int[] stripCounts = new int[m-1]; //带数目数组

```

384

```

42     for (int i = 0; i < m-1; i++) stripCounts[i] = 2*n;
43
44     GeometryInfo gi = //创建GeometryInfo几何数据对象
45     new GeometryInfo(GeometryInfo.TRIANGLE_STRIP_ARRAY);
46     gi.setCoordinates(pts); //设置顶点数组
47     gi.setCoordinateIndices(coords); //设置顶点坐标索引数组
48     gi.setStripCounts(stripCounts); //设置带数目数组
49
50     NormalGenerator ng = new NormalGenerator();
51     ng.generateNormals(gi); //生成法向量
52     this.setGeometry(gi.getGeometryArray());
53 }
54 //根据参数t和四个控制点, 插值计算Bezier曲线上的一点
55 Point3d deCasteljau(double t, Point3d[] p) {
56     Point3d[] pt = {new Point3d(p[0]),
57     new Point3d(p[1]), new Point3d(p[2]), new Point3d(p[3])};
58     for (int i = 0; i < 3; i++) {
59         for (int j = 0; j < 3-i; j++) {
60             pt[j].interpolate(pt[j+1], t);
61         }
62     }
63     return pt[0];
64 }
65 }

```

程序清单12-4 TestBezierSurface.java

```

1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 //定义TestBezierSurface类, 继承自Applet类, 用于演示使用BezierSurface类
12 public class TestBezierSurface extends Applet {
13     public static void main(String[] args) {
14         new MainFrame(new TestBezierSurface(), 640, 480); //创建程序主窗口并设置大小
15     }
16     //重写Applet初始化方法
17     public void init() {
18         //创建Canvas3D画布对象
19         GraphicsConfiguration gc =
20             SimpleUniverse.getPreferredConfiguration();
21         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
22         setLayout(new BorderLayout()); //设置布局管理器
23         add(cv, BorderLayout.CENTER);
24         BranchGroup bg = createSceneGraph(); //创建场景图分支
25         bg.compile();
26         SimpleUniverse su = new SimpleUniverse(cv); //创建设置SimpleUniverse对象
27         su.getViewingPlatform().setNominalViewingTransform();

```



```
28     su.addBranchGraph(bg);
29 }
30 //生成BranchGroup对象的私有方法, 创建场景图分支
31 private BranchGroup createSceneGraph() {
32     BranchGroup root = new BranchGroup(); //分支根节点
33     TransformGroup spin = new TransformGroup(); //几何变换组节点,
34     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
35     root.addChild(spin);
36     //定义曲面的控制点
37     Point3d[][] ctrlPts = new Point3d[4][4];
38     for (int i = 0; i < 4; i++) {
39         for (int j = 0; j < 4; j++) {
40             ctrlPts[i][j] = new Point3d(2-i, 3*(Math.random()-0.5), j-2);
41         }
42     }
43     Shape3D shape = new BezierSurface(ctrlPts); //创建曲面的几何形状对象
44     Appearance ap = new Appearance(); //创建外观对象
45     ap.setMaterial(new Material()); //材质属性
46     shape.setAppearance(ap);
47     Transform3D tr = new Transform3D(); //几何变换
48     tr.setScale(0.25); //缩放
49     TransformGroup tg = new TransformGroup(tr); //几何变换组节点
50     spin.addChild(tg);
51     tg.addChild(shape);
52     //旋转
53     Alpha alpha = new Alpha(-1, 10000);
54     RotationInterpolator rotator =
55     new RotationInterpolator(alpha, spin);
56     BoundingSphere bounds = new BoundingSphere(); //球体作用范围边界对象
57     rotator.setSchedulingBounds(bounds);
58     spin.addChild(rotator);
59     //设置背景和光照
60     Background background = new Background(1f, 1f, 1f);
61     background.setApplicationBounds(bounds);
62     root.addChild(background);
63     AmbientLight light = new AmbientLight(true,
64     new Color3f(Color.red)); //添加红色环境光源
65     light.setInfluencingBounds(bounds);
66     root.addChild(light);
67     PointLight ptlight = new PointLight
68     (new Color3f(Color.lightGray),
69     new Point3f(1f,1f,1f), new Point3f(1f,0f,0f)); //添加浅灰色点光源
70     ptlight.setInfluencingBounds(bounds);
71     root.addChild(ptlight);
72     return root;
73 }
74 }
```

386

BezierSurface类封装了一个双三次Bézier曲面, 它通过计算表面上的 $m \times n$ 个点, 并形成一个多边形网格, 来实现Bézier曲面。表面上的点的计算基于deCasteljau算法, 三次Bézier曲面的16个控制点保存在2D数组参数ctrlPts中。为了计算表面上参数值为 (u, v) 的点, 首先分别由四组控制点ctrlPts[k], $k = 0, 1, 2, 3$, 定义四条Bézier曲线, 并计算这四条曲线在 u 处的点。计算得到的四个点存放在数组p中, 然后以此四个点为控制点, 定义一条新的Bézier曲线, 并求此曲线在

v 处的点，最后得到的点，就是此Bézier曲面在 (u, v) 处的点。

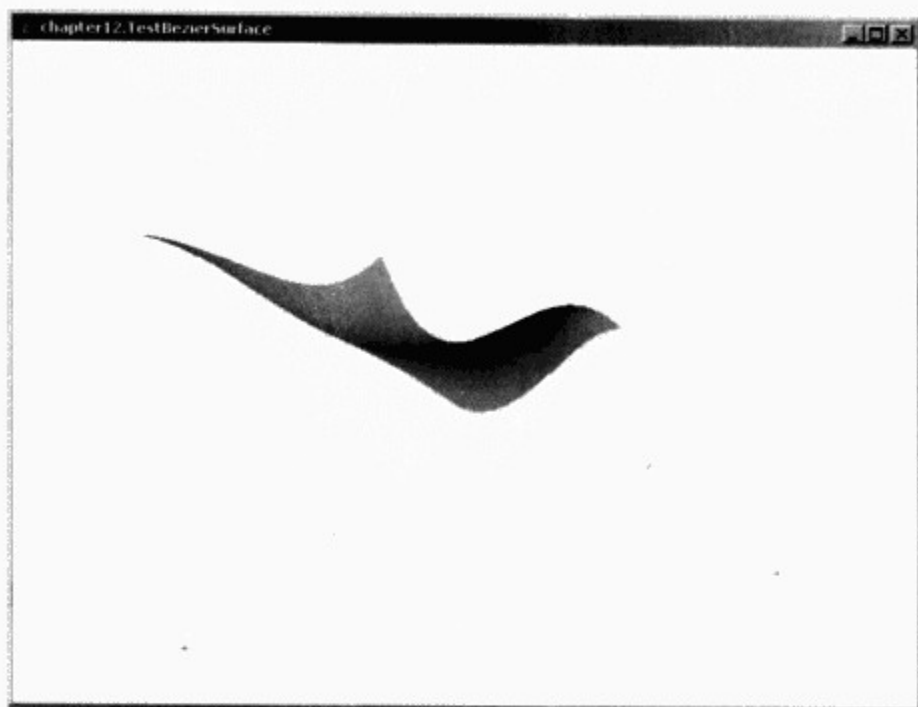


图12-3 一个双三次Bézier曲面

deCasteljau方法（第55行）根据所给参数和四个控制点，来计算Bézier曲线上的一个点。deCasteljau算法所需的线性插值计算，是直接调用Point3d类已有的interpolate方法（第60行）来进行的。

计算得到的曲面上的点，作为顶点存放在一个GeometryInfo类对象中，此GeometryInfo对象是以TRIANGLE_STRIP_ARRAY模式创建得到的，曲面的法向量则用一个NormalGenerator类对象自动生成。

TestBezierSurface类中的测试程序，基于随机生成数创建了16个控制点，并用BezierSurface类来产生可视对象的形状。该程序使用了光照，场景中放置了两个光源，程序还用了一个旋转插值器来旋转所生成的表面。

387

12.3.2 犹他茶壶

“犹他茶壶”是计算机图形学所创建的最富盛名的对象之一，它是由一组Bézier曲面片定义的。如程序清单12-5所示，可以用Bézier曲面类来显示这样一个茶壶（如图12-4所示）。

程序清单12-5 Teapot.java

```
1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import java.applet.*;
10 import com.sun.j3d.utils.applet.MainFrame;
11 import java.net.URL;
12 import java.io.*;
13 import java.util.StringTokenizer;
14 //定义Teapot对象，继承自Applet对象，演示一个由Bezier曲面构造的茶壶
15 public class Teapot extends Applet {
```



```
16 public static void main(String[] args) {
17     new MainFrame(new Teapot(), 640, 480); //创建主窗口并设置大小
18 }
19 //重写Applet初始化方法
20 public void init() {
21     //创建Canvas3D画布对象
22     GraphicsConfiguration gc =
23         SimpleUniverse.getPreferredConfiguration();
24     Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
25     setLayout(new BorderLayout()); //设置布局管理器
26     add(cv, BorderLayout.CENTER);
27     BranchGroup bg = createSceneGraph(); //创建场景图分支
28     bg.compile();
29     SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
30     su.getViewingPlatform().setNominalViewingTransform();
31     su.addBranchGraph(bg);
32 }
33 //生成BranchGroup对象, 创建场景图分支
34 private BranchGroup createSceneGraph() {
35     BranchGroup root = new BranchGroup(); //分支根节点
36     TransformGroup spin = new TransformGroup(); //几何变换组节点
37     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
38     root.addChild(spin);
39     //茶壶的几何数据
40     int n = 0;
41     int[][] idx = null;
42     int np = 0;
43     Point3d[] pts = null;
44     URL url =
45         getClass().getClassLoader().getResource("images/teapot");
46     try { // 读取茶壶的几何数据
47         BufferedReader br = new BufferedReader
48             (new InputStreamReader(url.openStream()));
49         String line = br.readLine();
50         n = Integer.parseInt(line); //字符串到整数的转换
51         idx = new int[n][16];
52         for (int i = 0; i < n; i++) {
53             line = br.readLine(); //读取一行, 并把一行的字符串分段
54             StringTokenizer st = new StringTokenizer(line, ", \n");
55             for (int j = 0; j < 16; j++) {
56                 idx[i][j] = Integer.parseInt(st.nextToken()); //每段转换到一个整数
57             }
58         }
59         line = br.readLine(); //读取一行
60         np = Integer.parseInt(line);
61         pts = new Point3d[np];
62         for (int i = 0; i < np; i++) {
63             line = br.readLine();
64             StringTokenizer st = new StringTokenizer(line, ", \n"); //字符串分段
65             double x = Double.parseDouble(st.nextToken());
66             double y = Double.parseDouble(st.nextToken());
67             double z = Double.parseDouble(st.nextToken());
68             pts[i] = new Point3d(x, y, z); //将各分段转为double数值, 并存储为点坐标
69         }
70     }
71 }
```

```

70     br.close();
71 } catch (IOException ex) {
72     ex.printStackTrace();
73 }
74
75 //构造曲面
76 Appearance ap = new Appearance();//外观对象
77 ap.setMaterial(new Material()); //材质属性
78 Transform3D tr = new Transform3D();//几何变换
79 tr.rotX(-Math.PI*0.5); //旋转变换
80 tr.setScale(0.25); //缩放变换
81 tr.setTranslation(new Vector3d(0,-0.5,0)); //平移变换
82 TransformGroup tg = new TransformGroup(tr); //几何变换组节点
83 spin.addChild(tg);
84 Point3d[][] ctrlPts = new Point3d[4][4]; //存储控制点
85 for (int k = 0; k < n; k++) {
86     for (int i = 0; i < 4; i++) {
87         for (int j = 0; j < 4; j++) {
88             ctrlPts[i][j] = pts[idx[k][i+4*j]-1];
89         }
90     }
91     Shape3D shape = new BezierSurface(ctrlPts); //构造Bezier曲面
92     shape.setAppearance(ap); //设置外观对象
93     tg.addChild(shape);
94 }
95 // 旋转
96 Alpha alpha = new Alpha(-1, 10000);
97 RotationInterpolator rotator =
98     new RotationInterpolator(alpha, spin);
99 BoundingSphere bounds = new BoundingSphere(); //球体作用范围边界对象
100 bounds.setRadius(10);
101 rotator.setSchedulingBounds(bounds);
102 spin.addChild(rotator);
103 //设置背景和光源
104 Background background = new Background(1f, 1f, 1f); //设置背景颜色
105 background.setApplicationBounds(bounds); //设置作用范围边界
106 root.addChild(background); //设置背景对象
107 AmbientLight light = new AmbientLight(true,
108     new Color3f(Color.white)); //添加白色环境光源
109 light.setInfluencingBounds(bounds); //设置作用范围边界
110 root.addChild(light);
111 PointLight ptlight = new PointLight(new Color3f(Color.white),
112     new Point3f(0.7f,1.8f,1.8f), new Point3f(1f,0.2f,0f)); //添加白色点光源
113 ptlight.setInfluencingBounds(bounds); // 设置作用范围边界
114 root.addChild(ptlight);
115 return root;
116 }
117 }

```

生成的该茶壶对象的相关数据，存储在一个文本文件“teapot”中，这些数据包括所有控制点的坐标及其索引号。程序从该文件读取相关数据，并用BezierSurface类构造出茶壶的各个面。

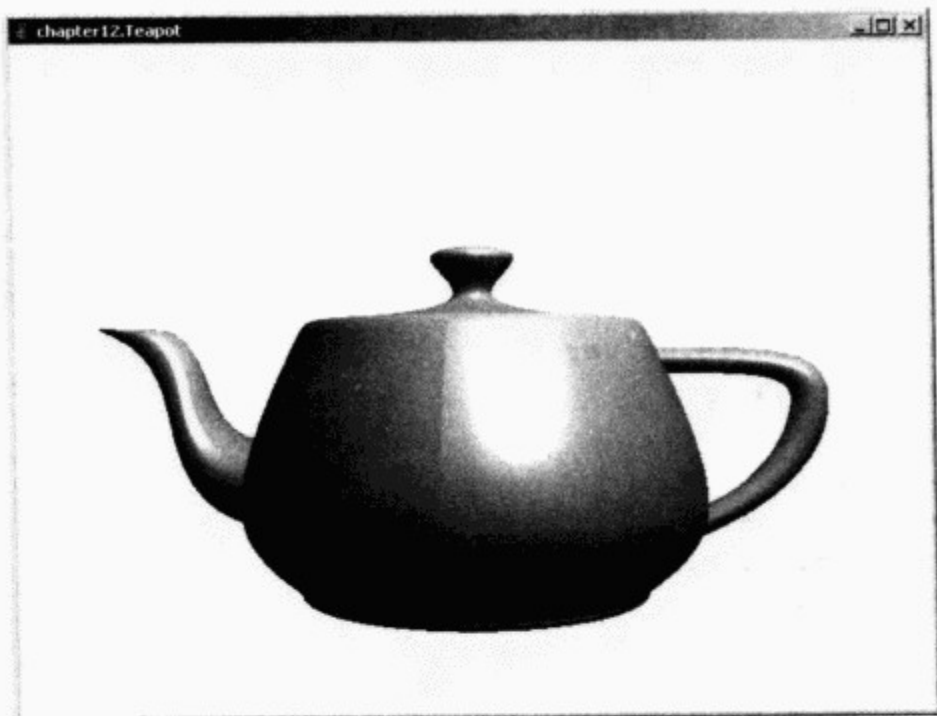


图12-4 犹他茶壶

12.4 声音

声音已经成为多媒体应用中的重要组成部分。虽然声音本质上并非一种图形对象，但是在游戏等计算机图形应用中，经常把声音与图形关联在一起。通过Sound类及其子类，Java 3D支持在场景图中嵌入声音。Sound类对象属于场景图中的叶节点，图12-5给出了Sound的类层次关系。

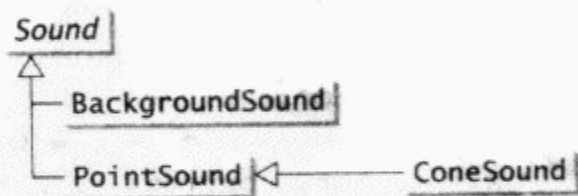


图12-5 Sound类及其子类

Sound类与Light类相似。就像AmbientLight类一样，BackgroundSound类对象也是均匀地分布在空间中。而PointSound类则有确定的位置，并且可能有衰减效果，就如PointLight类一样。ConeSound类对象的声音效果限制在一个锥形区域内，就像SpotLight类一样。

要在Java 3D程序中添加声音（add sound）对象，通常需要如下几个步骤：

- 创建一个AudioDevice类对象。如果场景图使用了SimpleUniverse对象，那么只要调用Viewer对象的createAudioDevice()方法，就可以方便地生成AudioDevice类对象：

```
su.getViewer().createAudioDevice();
```

- 创建一个MediaContainer类对象来保存声音数据。例如：

```
MediaContainer mc = new MediaContainer(url);
```

- 用声音数据创建一个Sound类对象，并将它加入到场景图中。例如：

```
BackgroundSound sound = new BackgroundSound();
sound.setSoundData(mc);
sound.setSchedulingBounds(bounds);
root.addChild(sound);
```

和其他环境类节点一样，声音节点需要设置它的作用范围边界。

程序清单12-6示例了如何在Java 3D中使用声音。这个例子在天空背景中显示一只海鸥（如图12-6）。通过鼠标操作，可以对这只海鸥进行旋转、平移和缩放，海鸥所在的位置将持续地播放鸟叫声。如果海鸥向左飞翔，则叫声也会从左侧发出，如果海鸥移向远处，则叫声会减弱。

程序清单12-6 Sound3D.java

```
1 package chapter12;
2
3 import java.net.URL;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.awt.image.*;
7 import com.sun.j3d.utils.universe.*;
8 import javax.media.j3d.*;
9 import javax.vecmath.*;
10 import java.io.*;
11 import javax.imageio.*;
12 import com.sun.j3d.utils.behaviors.mouse.*;
13 import chapter10.GullCG;
14 import java.applet.*;
15 import com.sun.j3d.utils.applet.MainFrame;
16 //定义Sound3D类,继承自Applet类,演示3D场景中的声音效果
17 public class Sound3D extends Applet {
18     public static void main(String[] args) {
19         new MainFrame(new Sound3D(), 640, 480); //创建主窗口并设置大小
20     }
21     //重写Applet初始化方法
22     public void init() {
23         //创建Canvas3D画布对象
24         GraphicsConfiguration gc =
25             SimpleUniverse.getPreferredConfiguration();
26         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
27         setLayout(new BorderLayout()); //设置布局管理器
28         add(cv, BorderLayout.CENTER);
29         SimpleUniverse su = new SimpleUniverse(cv); //创建SimpleUniverse
30         AudioDevice audioDev = su.getViewer().createAudioDevice(); //设置声音设备
31         BranchGroup bg = createSceneGraph(); //创建场景图分支
32         bg.compile();
33         su.getViewingPlatform().setNominalViewingTransform();
34         su.addBranchGraph(bg);
35     }
36     //生成BranchGroup对象,创建场景图
37     public BranchGroup createSceneGraph() {
38         //场景图分支根节点
39         BranchGroup objRoot = new BranchGroup();
40         Transform3D trans = new Transform3D(); //几何变换
41         trans.setTranslation(new Vector3d(Math.random()-0.5,
42             Math.random()-0.5, Math.random()-0.5)); //平移变换
43         trans.setScale(0.3); //缩放变换
44         TransformGroup objTrans = new TransformGroup(trans); //几何变换组节点
45         objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
46         objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
47         objRoot.addChild(objTrans);
48         //创建海鸥形体
49         Appearance ap = new Appearance(); //外观对象
50         ap.setMaterial(new Material()); //材质属性
51         Shape3D shape = new Shape3D(new GullCG(), ap);
52         objTrans.addChild(shape);
53         //行为
```

391


```
54     BoundingSphere bounds = //球体作用范围边界对象
55         new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
56     //设置鼠标旋转
57     MouseRotate rotator = new MouseRotate(objTrans);
58     rotator.setSchedulingBounds(bounds);
59     objRoot.addChild(rotator);
60     //设置鼠标平移
61     MouseTranslate translator = new MouseTranslate(objTrans);
62     translator.setSchedulingBounds(bounds);
63     objTrans.addChild(translator);
64     //设置鼠标缩放
65     MouseZoom zoom = new MouseZoom(objTrans);
66     zoom.setSchedulingBounds(bounds);
67     objTrans.addChild(zoom);
68     //创建声音对象
69     PointSound sound = new PointSound();
70     URL url =
71         this.getClass().getClassLoader().getResource("images/bird.au");//获取URL
72     MediaContainer mc = new MediaContainer(url);
73     sound.setSoundData(mc);
74     sound.setLoop(Sound.INFINITE_LOOPS);
75     sound.setInitialGain(1f);
76     sound.setEnable(true);
77     float[] distances = {1f, 20f};
78     float[] gains = {1f, 0.001f};
79     sound.setDistanceGain(distances, gains);
80     BoundingSphere soundBounds = //球体作用范围边界对象
81         new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
82     sound.setSchedulingBounds(soundBounds);
83     objTrans.addChild(sound);
84     //设置光照
85     AmbientLight light =
86         new AmbientLight(true, new Color3f(Color.blue)); //添加蓝色环境光源
87     light.setInfluencingBounds(bounds);
88     objRoot.addChild(light);
89     PointLight ptlight = new PointLight(new Color3f(Color.white),
90         new Point3f(0f,0f,2f), new Point3f(1f,0.3f,0f)); //添加白色点光源
91     ptlight.setInfluencingBounds(bounds);
92     objRoot.addChild(ptlight);
93     //根据图片设置背景
94     url = getClass().getClassLoader().getResource("images/bg.jpg");
95     BufferedImage bi = null;
96     try {
97         bi = ImageIO.read(url);
98     } catch (IOException ex) {
99         ex.printStackTrace();
100     }
101     ImageComponent2D image =
102         new ImageComponent2D(ImageComponent2D.FORMAT_RGB, bi);
103     Background background = new Background(image); //图像背景
104     background.setApplicationBounds(bounds);
105     objRoot.addChild(background);
106     return objRoot;
107 }
108 }
```

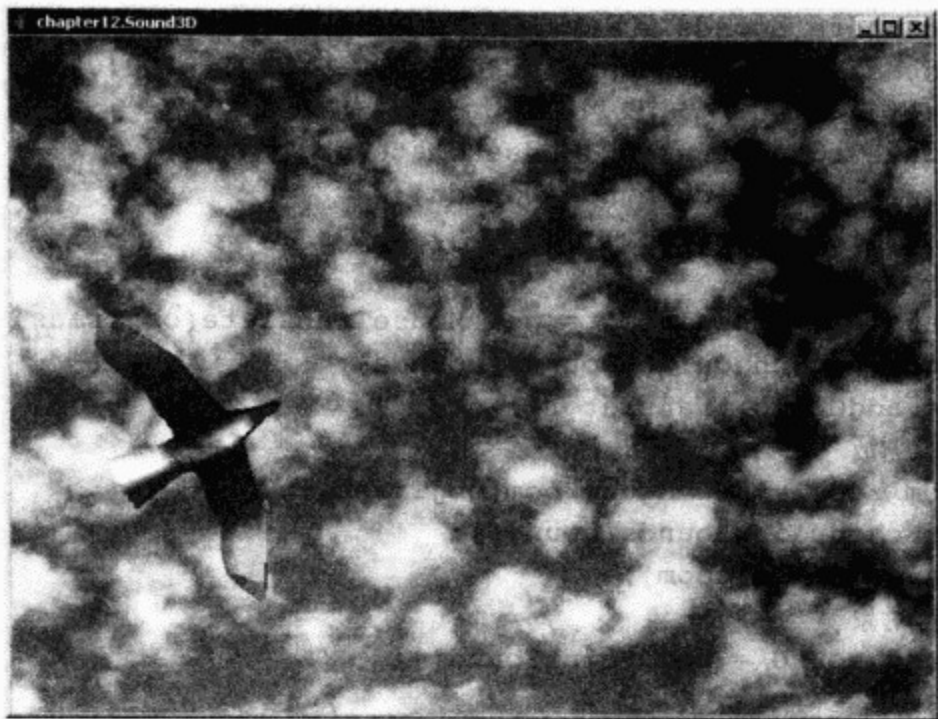


图12-6 和海鸥关联的点类声音对象

图12-7给出了嵌入声音节点的场景图，声音节点和海鸥形状节点接受相同的变换，该变换与三种鼠标操作行为联系在一起。当用户通过鼠标操作移动可视对象时，声音节点也跟着一起移动。

393

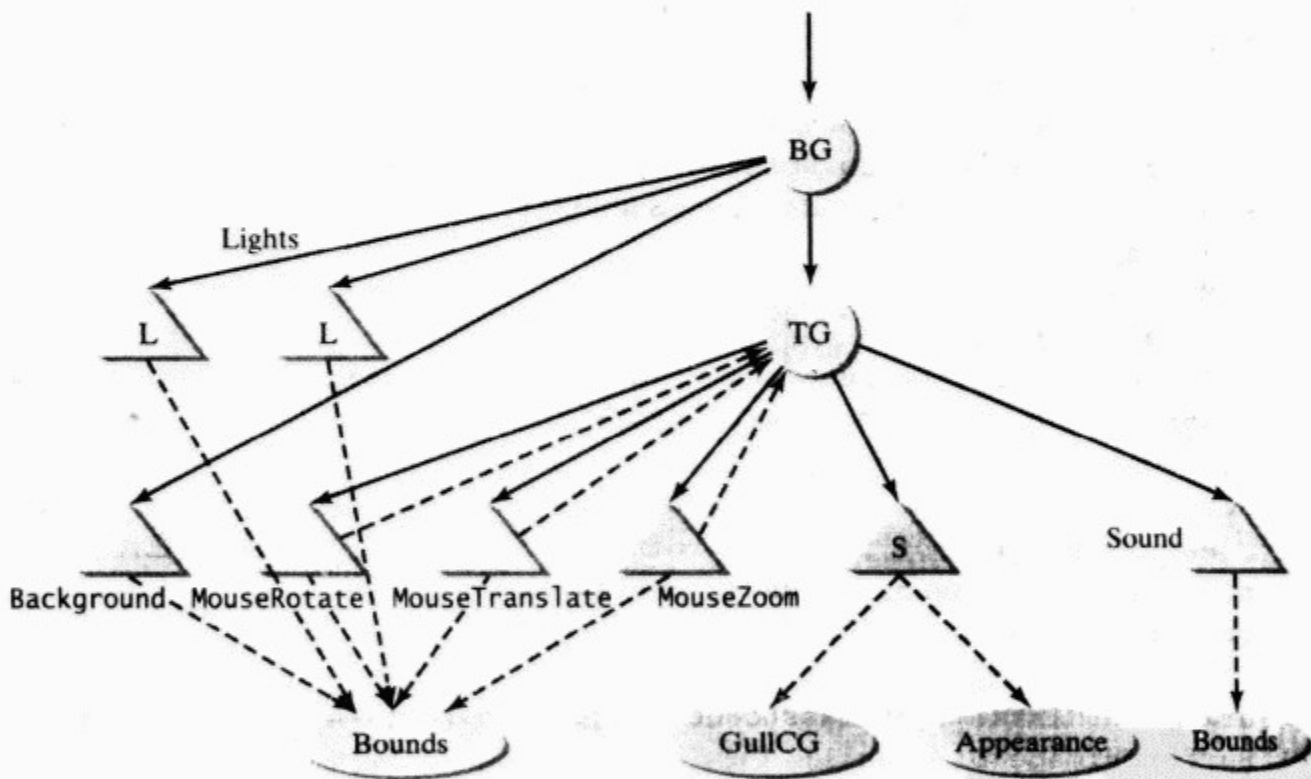


图12-7 声音示例的场景图

这个例子中的声音节点，是一个声音强度随距离衰减的PointSound类对象（第69行）。程序中定义了两个距离值和相应的声音增益值，当距离从1增长到20时，声音的增益从1衰减到0.001。

12.5 阴影

作为一个实时图像API，Java 3D只提供了局部光照选项。每个图形对象分别进行绘制，并不考虑它们之间的相互影响。例如，从一个对象反射的光，不会对其他对象有任何影响。当一个对象位于光源和另一个对象之间，部分地挡住了光源的光时，也不会对绘制效果有任何影响。

因此,这种局部光照模型并不会自动地产生阴影效果。

在场景中产生有真实感的阴影,是一种非常复杂的任务。场景中存在多个光源,不同图形对象之间的相互关系以及光源和图形对象的特征等,都是生成真实感阴影所需要考虑的因素。应用全局光照模型,可以得到较好的阴影效果,但全局光照技术计算开销巨大,通常难以用于实时图形绘制。本节将介绍一种生成人工阴影 (artificial shadows) 的方法,该方法用多边形对象来模拟阴影效果。虽然这种方法受到很多的限制,但它的效率很高。

假设将阴影投射到一个平面上,并且只考虑单个点光源。从光源位置出发,把图形对象的顶点投影到这个平面上,就可以得到用于模拟阴影的多边形顶点。图12-8演示了阴影多边形的构造过程。

这个转换过程是一个投影过程,类似于在视图上的投影。光源位置对应于观察者,唯一的区别在于,这一映射实际上是把所有的点映射到平面上,因此丢失了深度信息。如果光源位于坐标系原点,且投影平面为 $x = -d$,则这种标准投影形式所对应的转换矩阵为:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}$$

针对光源位置和阴影平面的一般配置,可以构造一个仿射变换 U ,用于将配置映射为标准形式。Transform3D类的lookAt方法可以产生这样的投影转换,最终的投影矩阵可以表示为:

$$U^{-1}PU$$

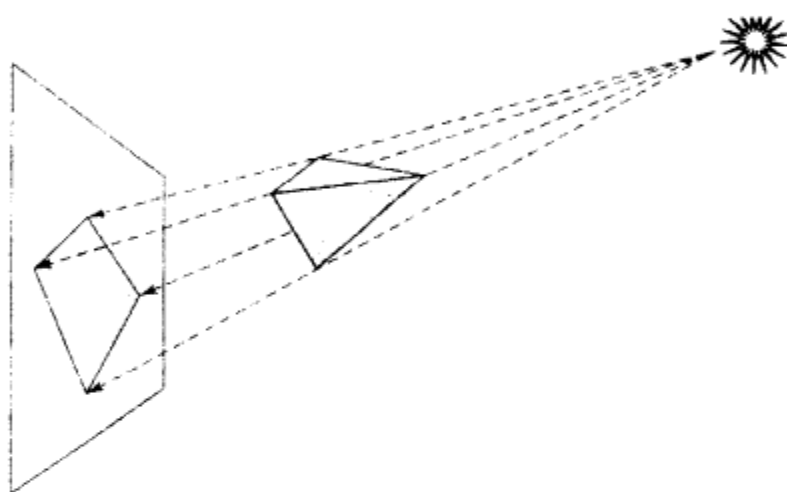


图12-8 用投影生成阴影

使用这个投影,任何几何体的顶点都可以映射到投影平面上,形成阴影多边形的顶点。

程序清单12-7实现了阴影的生成,该场景中包含一面石墙和一个十二面体,一个点光源照射在十二面体上,并在墙上投下阴影(如图12-9所示)。

程序清单12-7 Shadow.java

```
1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.awt.image.*;
7 import java.net.URL;
8 import java.io.*;
9 import javax.imageio.*;
10 import javax.swing.*;
11 import javax.media.j3d.*;
12 import com.sun.j3d.utils.universe.*;
13 import com.sun.j3d.utils.geometry.*;
14 import chapter6.Dodecahedron;
15 import java.applet.*;
16 import com.sun.j3d.utils.applet.MainFrame;
17 //定义Shadow类,继承自Applet类,演示物体在一个墙面上的阴影效果
18 public class Shadow extends Applet {
```

```

19 public static void main(String[] args) {
20     new MainFrame(new Shadow(), 640, 480); //创建主窗口并设置大小
21 }
22 //重写Applet初始化方法
23 public void init() {
24     //创建Canvas3D画布对象
25     GraphicsConfiguration gc =
395     SimpleUniverse.getPreferredConfiguration();
26     Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
27     setLayout(new BorderLayout()); //设置布局管理器
28     add(cv, BorderLayout.CENTER);
29     BranchGroup bg = createSceneGraph(); //创建场景图分支
30     bg.compile();
31     SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
32     su.getViewingPlatform().setNominalViewingTransform();
33     su.addBranchGraph(bg);
34 }
35 //生成BranchGroup的私有方法, 创建场景图
36 private BranchGroup createSceneGraph() {
37     BranchGroup root = new BranchGroup(); //定义根节点
38     //创建图形对象
39     Appearance ap = new Appearance(); //外观对象
40     ap.setMaterial(new Material()); //材质属性
41     Shape3D shape = new Dodecahedron(); //创建正十二面体
42     shape.setAppearance(ap);
43     GeometryArray geom = (GeometryArray)shape.getGeometry();
44     //几何变换
45     Transform3D tr = new Transform3D();
46     tr.rotY(-0.2); //旋转变换
47     tr.setScale(0.2); //缩放变换
48     TransformGroup tg = new TransformGroup(tr); //几何变换组节点
49     root.addChild(tg);
50     tg.addChild(shape);
51     BoundingSphere bounds =
52         new BoundingSphere(new Point3d(0,0,0),100); // 球体作用范围边界对象
53     //设置光源和背景对象
54     Background background = new Background(1.0f, 1.0f, 1.0f); //设置背景颜色
55     background.setApplicationBounds(bounds);
56     root.addChild(background); //设置背景对象
57     AmbientLight light =
58         new AmbientLight(true, new Color3f(Color.red)); //添加红色环境光源
59     light.setInfluencingBounds(bounds); //设置作用范围边界
60     root.addChild(light);
61     Point3f lightPos = new Point3f(10f, 3f, 1f);
62     PointLight ptlight = new PointLight(new Color3f(Color.green),
63         lightPos, new Point3f(1f, 0f, 0f)); //添加绿色点光源
64     ptlight.setInfluencingBounds(bounds); //设置作用范围边界
65     tg.addChild(ptlight);
66     //墙对象
67     Shape3D wall = createWall();
68     tg.addChild(wall);
69     //阴影对象
70     GeometryArray shadow = createShadow(geom,
71         lightPos, new Point3f(-2f, 3f, 1f));
72 }

```



```

73     ap = new Appearance();//外观对象
74     ColoringAttributes colorAttr =
75         new ColoringAttributes(0.1f, 0.1f, 0.1f,
76             ColoringAttributes.FASTEST);
77     ap.setColoringAttributes(colorAttr);//颜色属性
78     TransparencyAttributes transAttr = new TransparencyAttributes(
79         TransparencyAttributes.BLENDED,0.35f);
80     ap.setTransparencyAttributes(transAttr);//透明属性
81     PolygonAttributes polyAttr = new PolygonAttributes();
82     polyAttr.setCullFace(PolygonAttributes.CULL_NONE);
83     ap.setPolygonAttributes(polyAttr); //多边形绘制属性
84     shape = new Shape3D(shadow, ap);
85     tg.addChild(shape);
86     return root;
87 }
88 //创建墙体多边形对象
89 private Shape3D createWall() {
90     URL url =
91         getClass().getClassLoader().getResource("images/stone.jpg");
92     BufferedImage bi = null;
93     try {
94         bi = ImageIO.read(url);//读取图像
95     } catch (IOException ex) {
96         ex.printStackTrace();
97     }
98     ImageComponent2D image =
99         new ImageComponent2D(ImageComponent2D.FORMAT_RGB, bi);
100     Texture2D texture =
101         new Texture2D(Texture.BASE_LEVEL, Texture.RGBA,
102             image.getWidth(), image.getHeight());//基于图像构造纹理属性
103     texture.setImage(0, image);
104     texture.setEnabled(true);
105     texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
106     texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
107     Appearance appear = new Appearance(); //外观对象
108     appear.setTexture(texture); //设置纹理属性
109     QuadArray rect = new QuadArray(4, QuadArray.COORDINATES |
110         QuadArray.TEXTURE_COORDINATE_2);
111     rect.setCoordinate(0, new Point3d(-2,3,2));
112     rect.setCoordinate(1, new Point3d(-2,-3,2));
113     rect.setCoordinate(2, new Point3d(-2,-3,-3));
114     rect.setCoordinate(3, new Point3d(-2,3,-3));
115     rect.setTextureCoordinate(0,0, new TexCoord2f(0f, 0f));
116     rect.setTextureCoordinate(0,1, new TexCoord2f(0f, 1f));
117     rect.setTextureCoordinate(0,2, new TexCoord2f(1f, 1f));
118     rect.setTextureCoordinate(0,3, new TexCoord2f(1f, 0f));
119     return new Shape3D(rect, appear);
120 }
121 //创建阴影对象
122 private GeometryArray createShadow(GeometryArray ga, Point3f light,
123     Point3f plane) {
124     GeometryInfo gi = new GeometryInfo(ga);
125     gi.convertToIndexedTriangles();
126     IndexedTriangleArray ita =

```

396

```

127     (IndexedTriangleArray)gi.getIndexedGeometryArray();
128     Vector3f v = new Vector3f();
129     v.sub(plane, light);
130     double[] mat = new double[16]; //标准位置的投影矩阵
131     for (int i = 0; i < 16; i++) {
132         mat[i] = 0;
133     }
134     mat[0] = 1;
135     mat[5] = 1;
136     mat[10] = 1-0.001;
137     mat[14] = -1/v.length();
138     Transform3D proj = new Transform3D();
139     proj.set(mat);
140     Transform3D u = new Transform3D();
141     u.lookAt(new Point3d(light),
142     new Point3d(plane), new Vector3d(0,1,0)); //创建到标准位置的几何变换矩阵
143     proj.mul(u);
144     Transform3D tr = new Transform3D();
145     u.invert();
146     tr.mul(u, proj); // 按照公式计算最终的投影矩阵
147     int n = ita.getVertexCount();
148     int count = ita.getIndexCount();
149     IndexedTriangleArray shadow = new IndexedTriangleArray(n,
150     GeometryArray.COORDINATES, count);
151     for (int i = 0; i < n; i++) { //计算各点的投影
152         Point3d p = new Point3d();
153         ga.getCoordinate(i, p);
154         Vector4d v4 = new Vector4d(p);
155         v4.w = 1;
156         tr.transform(v4);
157         Point4d p4 = new Point4d(v4);
158         p.project(p4);
159         shadow.setCoordinate(i, p); //投影点作为阴影对象的几何数据
160     }
161     int[] indices = new int[count];
162     ita.getCoordinateIndices(0, indices);
163     shadow.setCoordinateIndices(0, indices);
164     return shadow;
165 }
166 }

```

该程序的场景图如图12-10所示，它包含一个带有映射纹理的矩形墙面、一个十二面体及其在墙上投下的阴影。程序使用了环境光和一个点光源，阴影的计算与点光源的位置有关。

createShadow方法（第123行）进行一些必要的计算，完成根据原对象的顶点投影创建阴影多边形的任务。所用的投影矩阵基于标准投影矩阵及其到标准位置的变换计算而来，到标准位置的变换可以通过lookAt方法加以实现。实际上该投影稍稍位于墙面之上，以避免阴影和墙体对象之间发生相互干扰。

阴影多边形对象的颜色为灰色，它是由ColoringAttributes类对象进行设置的。同时，这个多边形对象是半透明的，可以显示阴影下的一些墙体细节。阴影多边形的透明度，由一个TransparencyAttributes类对象进行设置（第73~83行）。

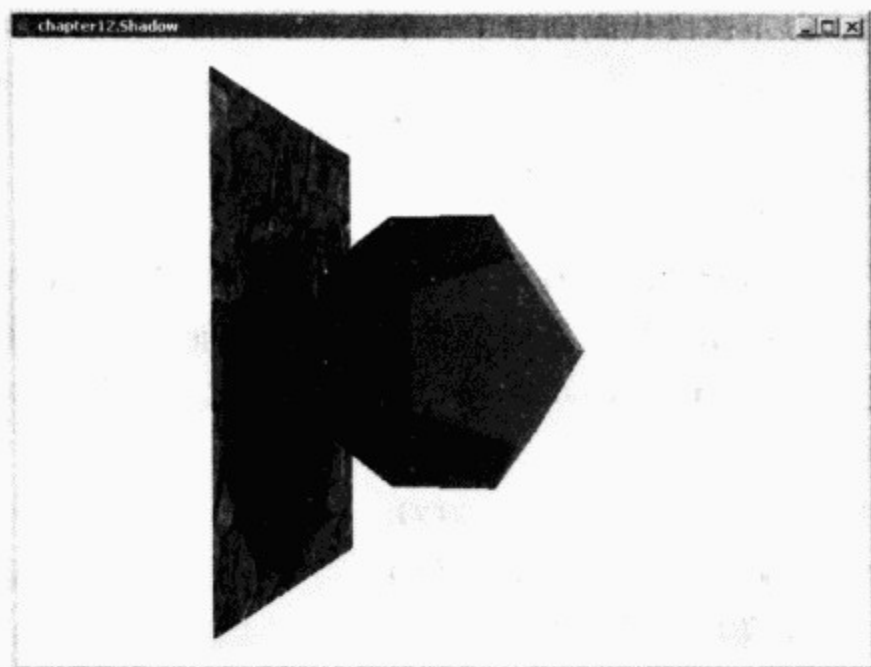


图12-9 利用对象的投影多边形，在墙面上生成阴影

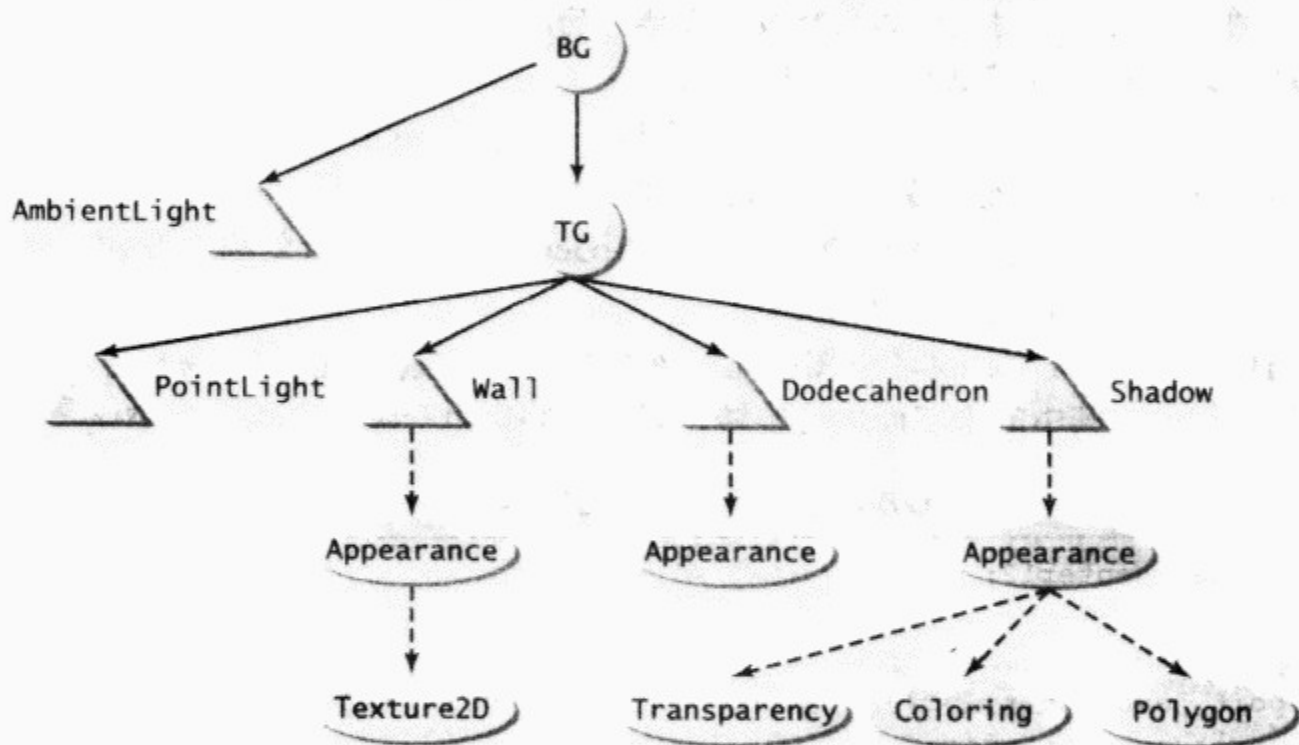


图12-10 阴影示例程序的场景图

12.6 几何变化

第10章和第11章所介绍的动画和行为对象，并不能修改活动场景图中的几何数据。Morph类对象可以把几个几何对象组合起来，但它只能修改其组合权重系数。在某些情况下，如果能够修改活动场景中实际的几何数据，那是很有用的。例如，上一节中介绍的人工阴影的例子，只能适用于静态的物体。如果十二面体相对于墙或光源发生移动，则阴影多边形对象也应该进行相应的变化，这就要求重新计算阴影多边形的各个顶点。简单地对阴影多边形施加和十二面体同样的一系列几何变换，并不能产生正确的结果，因为阴影多边形是通过十二面体进行一次投影转换后生成的。

Java 3D提供了一种机制，可以通过GeometryUpdater接口和GeometryArray类的BY_REFERENCE模式，来修改活动场景图的几何数据。在默认情况下，每个GeometryArray类对象都保存一份几何数据的拷贝，例如所有顶点的坐标。如果创建GeometryArray类对象时，在调用其构造函数时设置BY_REFERENCE标记，则该GeometryArray对象仅保存指向由用户提

供的数据的引用。例如，以下这段代码创建了一个基于引用的三角形几何数据：

```
GeometryArray geom = new TriangleArray(3, GeometryArray.BY_REFERENCE
                                     | GeometryArray.COORDINATES);
float[] coords = {1, 0, 0, 0, 1, 0, 0, 0, 1};
geom.setCoordRefFloat(coords);
```

数组coords包含了该几何对象的所有顶点坐标，并由一个TriangleArray类对象加以引用。

399 虽然已经为基于引用的几何数据提供了存储位置，但仍然不能通过它直接修改活动场景图的内容。正确的做法应该是实现GeometryUpdater接口，这个接口定义了如下一个方法，对于更新几何对象来说是必须要实现的：

```
public void updateData(Geometry geometry)
```

参数geometry提供了待修改几何对象的引用手段，这个方法是由系统进行调用的。此外，也可以调用GeometryArray对象的同名函数，来激活几何数据的更新：

```
public void updateData(GeometryUpdater updater)
```

调用上述方法时，需要传入一个自定义的GeometryUpdater实例参数。

总之，通过行为实现对几何体的修改，需要经历以下几个步骤：

- 以BY_REFERENCE模式创建几何对象。
- 编写一个实现GeometryUpdater接口的类，实现GeometryUpdater接口的updateData (Geometry)方法，从而完成对几何数据的修改。
- 实现一个Behavior类。在恰当的时候，调用GeometryArray对象的updateData (GeometryUpdater)方法，完成几何数据的更新。

程序清单12-8示例了对活动场景中几何数据的修改，该程序创建的场景图与程序清单12-7的情况相似。但是，其中的十二面体会旋转，同时，阴影也会随之发生动态变化。

程序清单12-8 MovingShadow.java

```
1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.awt.image.*;
7 import java.net.URL;
8 import java.io.*;
9 import javax.imageio.*;
10 import javax.swing.*;
11 import javax.media.j3d.*;
12 import com.sun.j3d.utils.universe.*;
13 import com.sun.j3d.utils.geometry.*;
14 import chapter6.Dodecahedron;
15 import java.applet.*;
16 import com.sun.j3d.utils.applet.MainFrame;
17 //定义MovingShadow类，继承自Applet类，演示运动正十二面体所产生的阴影
18 public class MovingShadow extends Applet {
19     public static void main(String[] args) {
20         new MainFrame(new MovingShadow(), 640, 480); //创建主窗口并设置大小
21     }
22     //声明变换节点及几何形体节点对象
23     private TransformGroup spin = null;
```



```
24 private Transform3D shadowProj = null;
25 private GeometryArray geom = null;
26 private GeometryArray shadowGeom = null;
27 private ShadowUpdater updater = null;
28 //重写Applet初始化方法
29 public void init() {
30     //创建Canvas3D画布对象
31     GraphicsConfiguration gc =
32     SimpleUniverse.getPreferredConfiguration();
33     Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
34     setLayout(new BorderLayout()); //设置布局管理器
35     add(cv, BorderLayout.CENTER);
36     updater = new MovingShadow.ShadowUpdater(); //创建ShadowUpdater类对象
37     BranchGroup bg = createSceneGraph(); //创建场景图分支
38     bg.compile();
39     SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
40     su.getViewingPlatform().setNominalViewingTransform();
41     su.addBranchGraph(bg);
42 }
43 //生成BranchGroup的私有方法, 创建场景图
44 private BranchGroup createSceneGraph() {
45     BranchGroup root = new BranchGroup(); //场景图分支根节点
46     //构造图形对象
47     Appearance ap = new Appearance(); //外观对象
48     ap.setMaterial(new Material()); //材质属性
49     Shape3D shape = new Dodecahedron(); //创建正十二面体
50     shape.setAppearance(ap);
51     geom = (GeometryArray)shape.getGeometry();
52     geom.setCapability(GeometryArray.ALLOW_COORDINATE_READ);
53     //几何变换
54     Transform3D tr = new Transform3D();
55     tr.rotY(-0.2);
56     tr.setScale(0.2);
57     TransformGroup tg = new TransformGroup(tr);
58     root.addChild(tg);
59     spin = new TransformGroup();
60     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
61     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
62     tg.addChild(spin);
63     spin.addChild(shape);
64     //旋转
65     Alpha alpha = new Alpha(-1, 8000);
66     RotationInterpolator rotator =
67     new RotationInterpolator(alpha, spin);
68     BoundingSphere bounds = new BoundingSphere();
69     rotator.setSchedulingBounds(bounds);
70     spin.addChild(rotator);
71     //设置光源和背景对象
72     Background background = new Background(1.0f, 1.0f, 1.0f);
73     background.setApplicationBounds(bounds);
74     root.addChild(background);
75     AmbientLight light =
76     new AmbientLight(true, new Color3f(Color.red)); //添加红色环境光源
77     light.setInfluencingBounds(bounds);
```

400

```

78     root.addChild(light);
79     Point3f lightPos = new Point3f(10f, 3f, 1f);
80     PointLight ptlight = new PointLight(new Color3f(Color.green),
81     lightPos, new Point3f(1f, 0f, 0f)); //添加绿色点光源
82     ptlight.setInfluencingBounds(bounds);
83     tg.addChild(ptlight);
84     //构造墙对象
85     Shape3D wall = createWall();
86     tg.addChild(wall);
87     //构造阴影对象
88     shadowGeom =
89     createShadow(geom, lightPos, new Point3f(-2f, 3f, 1f));
90     ap = new Appearance();
91     ColoringAttributes colorAttr =
92     new ColoringAttributes(0.1f, 0.1f, 0.1f,
93     ColoringAttributes.FASTEST);
94     ap.setColoringAttributes(colorAttr); //颜色属性
95     TransparencyAttributes transAttr = new TransparencyAttributes(
96     TransparencyAttributes.BLENDED, 0.35f); //透明属性
97     ap.setTransparencyAttributes(transAttr);
98     PolygonAttributes polyAttr = new PolygonAttributes();
99     polyAttr.setCullFace(PolygonAttributes.CULL_NONE);
100    ap.setPolygonAttributes(polyAttr); //多边形绘制属性
101    shape = new Shape3D(shadowGeom, ap);
102    tg.addChild(shape);
103    //对阴影的更新
104    ShadowBehavior sb = new MovingShadow.ShadowBehavior();
105    sb.setSchedulingBounds(bounds);
106    tg.addChild(sb);
107    return root;
108 }
109 //创建墙对象
110 private Shape3D createWall() {
111     URL url =
112     getClass().getClassLoader().getResource("images/stone.jpg");
113     BufferedImage bi = null;
114     try {
115         bi = ImageIO.read(url); //读入图像
116     } catch (IOException ex) {
117         ex.printStackTrace();
118     }
119     ImageComponent2D image =
120     new ImageComponent2D(ImageComponent2D.FORMAT_RGB, bi);
121     Texture2D texture =
122     new Texture2D(Texture.BASE_LEVEL, Texture.RGBA,
123     image.getWidth(), image.getHeight()); //基于图像构造纹理属性
124     texture.setImage(0, image);
125     texture.setEnable(true);
126     texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
127     texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
128     Appearance appear = new Appearance(); //外观对象
129     appear.setTexture(texture); //纹理属性
130     QuadArray rect = new QuadArray
131     (4, QuadArray.COORDINATES | QuadArray.TEXTURE_COORDINATE_2);

```



```
132 rect.setCoordinate(0, new Point3d(-2,3,2));
133 rect.setCoordinate(1, new Point3d(-2,-3,2));
134 rect.setCoordinate(2, new Point3d(-2,-3,-3));
135 rect.setCoordinate(3, new Point3d(-2,3,-3));
136 rect.setTextureCoordinate(0,0, new TexCoord2f(0f, 0f));
137 rect.setTextureCoordinate(0,1, new TexCoord2f(0f, 1f));
138 rect.setTextureCoordinate(0,2, new TexCoord2f(1f, 1f));
139 rect.setTextureCoordinate(0,3, new TexCoord2f(1f, 0f));
140 return new Shape3D(rect, appear);
141 }
142 //创建阴影对象
143 private GeometryArray createShadow
144     (GeometryArray ga, Point3f light, Point3f plane) {
145     GeometryInfo gi = new GeometryInfo(ga);
146     gi.convertToIndexedTriangles();
147     IndexedTriangleArray ita =
148         (IndexedTriangleArray)gi.getIndexedGeometryArray();
149     Vector3f v = new Vector3f();
150     v.sub(plane, light);
151     double[] mat = new double[16]; //标准位置的投影矩阵
152     for (int i = 0; i < 16; i++) {
153         mat[i] = 0;
154     }
155     mat[0] = 1;
156     mat[5] = 1;
157     mat[10] = 1-0.001;
158     mat[14] = -1/v.length();
159     Transform3D proj = new Transform3D(); //几何变换
160     proj.set(mat);
161     Transform3D u = new Transform3D();
162     u.lookAt(new Point3d(light), //创建到标准位置的几何变换矩阵
163         new Point3d(plane), new Vector3d(0,1,0));
164     proj.mul(u);
165     shadowProj = new Transform3D(); //最终投影矩阵
166     u.invert();
167     shadowProj.mul(u, proj); //按照公式计算最终的投影矩阵
168     int n = ita.getVertexCount();
169     int count = ita.getIndexCount();
170     IndexedTriangleArray shadow = new IndexedTriangleArray(n,
171         GeometryArray.COORDINATES | GeometryArray.BY_REFERENCE, count);
172     shadow.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
173     shadow.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
174     double[] vert = new double[3*n];
175     Point3d p = new Point3d();
176     for (int i = 0; i < n; i++) { //计算各点的投影
177         ga.getCoordinate(i, p);
178         Vector4d v4 = new Vector4d(p);
179         v4.w = 1;
180         shadowProj.transform(v4);
181         Point4d p4 = new Point4d(v4);
182         p.project(p4);
183         vert[3*i] = p.x;
184         vert[3*i+1] = p.y;
185         vert[3*i+2] = p.z;
```

```

186     }
187     shadow.setCoordRefDouble(vert); //投影点作为阴影对象的几何数据
188     int[] indices = new int[count];
189     ita.getCoordinateIndices(0, indices);
190     shadow.setCoordinateIndices(0, indices);
191     return shadow;
192 }
193 //定义ShadowUpdater类, 并实现GeometryUpdater接口
194 class ShadowUpdater implements GeometryUpdater {
195     public void updateData(Geometry geometry) {
196         double[] vert =
197             ((GeometryArray)geometry).getCoordRefDouble();
198         int n = vert.length/3;
199         Transform3D rot = new Transform3D();
200         spin.getTransform(rot);
201         Transform3D tr = new Transform3D(shadowProj);
202         tr.mul(rot);
203         Point3d p = new Point3d();
204         for (int i = 0; i < n; i++) { //重新计算各点的投影
205             geom.getCoordinate(i, p);
206             Vector4d v4 = new Vector4d(p);
207             v4.w = 1;
208             tr.transform(v4);
209             Point4d p4 = new Point4d(v4);
210             p.project(p4);
211             vert[3*i] = p.x;
212             vert[3*i+1] = p.y;
213             vert[3*i+2] = p.z;
214         }
215     }
216 }
217 定义ShadowBehavior类, 继承自Behavior类
218 class ShadowBehavior extends Behavior {
219     WakeupOnElapsedFrames wakeup = null;
220
221     public ShadowBehavior() {
222         wakeup = new WakeupOnElapsedFrames(0); //指定每一帧结束时被唤醒
223     }
224     //初始化方法
225     public void initialize() {
226         wakeupOn(wakeup); //初始化时首次唤醒
227     }
228     //触发对几何数据的修改
229     public void processStimulus(java.util.Enumeration enumeration) {
230         shadowGeom.updateData(updater);
231         wakeupOn(wakeup); //唤醒
232     }
233 }
234 }

```

该程序的场景图如图12-11所示, 与程序清单12-4的场景图(如图12-10所示)相比, 这个场景图中加入了几个新的节点。在场景中的十二面体节点之上, 添加了一个TransformGroup节点和一个RotationInterpolator节点, 对十二面体进行旋转。场景中还添加了一个行为节点, 用于

修改阴影对象的几何数据。

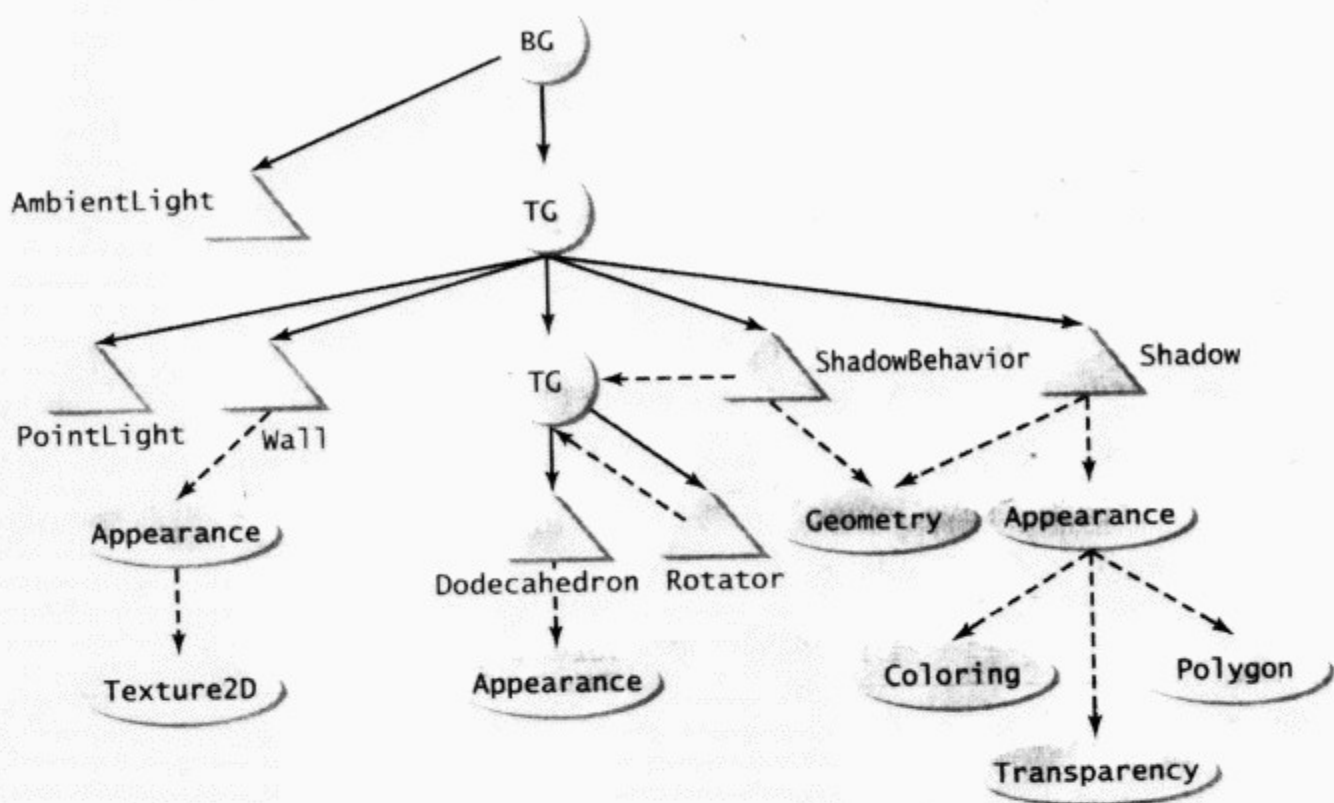


图12-11 移动阴影实例的场景图

404

createShadow方法（第143行）中用于创建阴影多边形所用的投影变换矩阵，与程序清单12-4所用矩阵是一样的。不过，在这个例子中，还以BY_REFERENCE模式创建了一个IndexedTriangleArray类对象（第170到171行），生成阴影的投影矩阵保存在成员变量shadowProj中。程序创建了一个float数组vert来存储阴影多边形各个顶点的坐标，阴影对象的几何数据保存在私有成员变量shadowGeom中。

程序中定义了一个内部类ShadowUpdater（第194行），用于对阴影多边形进行修改，这个内部类实现了GeometryUpdater接口。在它的updateData方法中，首先读取作用于十二面体对象的旋转变换矩阵，将其与shadowProj投影矩阵组合在一起，形成新的投影矩阵。这一新的投影矩阵随后被作用于十二面体的每一个顶点坐标，以获得阴影多边形的顶点坐标。构造函数中创建了一个ShadowUpdater类实例对象，并通过成员变量updater进行引用。

程序中还给出了一个作为内部类使用的自定义Behavior类ShadowBehavior（第218行），将它设置为在当前帧结束时被唤醒，这个类的processStimulus方法通过updater参数调用shadowGeom对象的updateData方法，以触发对几何数据的修改。场景图中还加入了一个ShadowBehavior类对象。

基于行为驱动的几何数据修改，使我们能够动态地改变阴影对象，这个程序显示了一个旋转的十二面体及其投在墙上的阴影，该阴影对象会随着十二面体的位置变化而变化。

12.7 离屏绘制

Java 3D中，绘制结果发送给Canvas3D类对象（画布对象）。通常情况下，Canvas3D对象都作为AWT组件显示在计算机屏幕上。但是在某些情况下，可能需要进行离屏绘制（off-screen rendering）。例如，如果需要保存或打印所绘制场景的图像，最好能把场景的绘制结果保存到基于内存的图像中，而不是把它绘制到计算机屏幕上。

Canvas3D类支持离屏绘制。如果要创建一个进行离屏绘制的Canvas3D对象，需要调用以下构造函数，并且把offScreen参数设为true：


```
public Canvas3D(GraphicsConfiguration gc, boolean offScreen)
```

要把场景绘制到离屏的画布对象上，需要把画布对象附属到View对象上：

```
view.addCanvas3D(canvas)
```

通过以下一系列Canvas3D对象方法的调用，可以把所绘制的场景保存下来：

```
ImageComponent2D buffer =
    new ImageComponent2D(ImageComponent.FORMAT_RGB, bImage);
canvas.setOffScreenBuffer(buffer);
canvas.renderOffScreenBuffer();
canvas.waitForOffScreenRendering();
bImage = offScreenCanvas.getOffScreenBuffer().getImage();
```

程序清单12-9示例了如何进行离屏绘制。屏幕上显示了一个旋转的十二面体（如图12-12）。窗口底部放置了一个标为“save image”的按钮，点击此按钮，将捕捉到当前显示的图像，用户可以选择把此图像保存为一个JPEG格式的文件。

程序清单12-9 OffScreen.java

```
1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.awt.image.*;
7 import java.net.URL;
8 import java.io.*;
9 import javax.imageio.*;
10 import javax.swing.*;
11 import javax.media.j3d.*;
12 import com.sun.j3d.utils.universe.*;
13 import com.sun.j3d.utils.geometry.*;
14 import chapter6.Dodecahedron;
15 定义OffScreen类，继承自Frame类，演示off-screen绘制
16 public class OffScreen extends Frame{
17     public static void main(String[] args) {
18         Frame frame = new OffScreen();//创建程序主窗口
19         frame.setTitle("Off Screen Rendering");//设置窗口标题栏
20         frame.setSize(640, 480);//设置窗口大小
21         frame.setVisible(true);//设置窗口可见
22     }
23     //声明类变量
24     private Canvas3D cv;
25     private Canvas3D offScreenCanvas;
26     private View view;
27
28     public OffScreen() {
29         WindowListener l = new WindowAdapter() { //定义窗口事件响应器
30             public void windowClosing(WindowEvent ev) { //当关闭窗口时退出程序
31                 System.exit(0);
32             }
33         };
34         this.addWindowListener(l); //添加窗口事件侦听器
35         GraphicsConfiguration gc =
36             SimpleUniverse.getPreferredConfiguration();
```



```
37 cv = new Canvas3D(gc); //创建Canvas3D画布对象
38 setLayout(new BorderLayout()); //设置布局管理器
39 add(cv, BorderLayout.CENTER);
40 BranchGroup bg = createSceneGraph(); //创建场景图分支
41 bg.compile();
42 SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
43 su.getViewingPlatform().setNominalViewingTransform();
44 su.addBranchGraph(bg);
45 //创建离屏画布对象
46 view = su.getViewer().getView();
47 offScreenCanvas = new Canvas3D(gc, true);
48 Screen3D sOn = cv.getScreen3D();
49 Screen3D sOff = offScreenCanvas.getScreen3D();
50 Dimension dim = sOn.getSize();
51 sOff.setSize(dim);
52 sOff.setPhysicalScreenWidth(sOn.getPhysicalScreenWidth());
53 sOff.setPhysicalScreenHeight(sOn.getPhysicalScreenHeight());
54 Point loc = cv.getLocationOnScreen();
55 offScreenCanvas.setOffScreenLocation(loc);
56 //添加按钮及事件侦听器
57 Button button = new Button("Save image");
58 add(button, BorderLayout.SOUTH);
59 button.addActionListener(new ActionListener() { //响应按钮事件
60     public void actionPerformed(ActionEvent ev) {
61         BufferedImage bi = capture(); //捕捉画面
62         save(bi);
63     }
64 });
65 }
66 //生成BranchGroup对象的私有方法, 创建场景图分支
67 private BranchGroup createSceneGraph() {
68     BranchGroup root = new BranchGroup(); //场景图分支根节点
69     TransformGroup spin = new TransformGroup();
70     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
71     root.addChild(spin);
72     //构造图形对象
73     Appearance ap = new Appearance(); //外观对象
74     ap.setMaterial(new Material()); //材质属性
75     Shape3D shape = new Dodecahedron(); //创建十二面体
76     shape.setAppearance(ap);
77     Transform3D tr = new Transform3D(); //几何变换
78     tr.setScale(0.25); //缩放变换
79     TransformGroup tg = new TransformGroup(tr); //几何变换组节点
80     spin.addChild(tg);
81     tg.addChild(shape);
82     Alpha alpha = new Alpha(-1, 10000);
83     RotationInterpolator rotator =
84     new RotationInterpolator(alpha, spin); //旋转对象
85     BoundingSphere bounds = new BoundingSphere();
86     rotator.setSchedulingBounds(bounds);
87     spin.addChild(rotator);
88     //设置背景和光源对象
89     URL url =
90     getClass().getClassLoader().getResource("images/bg.jpg");
```

406

```

91     BufferedImage bi = null;
92     try {
93         bi = ImageIO.read(url); //读取图像
94     } catch (IOException ex) {
95         ex.printStackTrace();
96     }
97     ImageComponent2D image =
98         new ImageComponent2D(ImageComponent2D.FORMAT_RGB, bi);
99     Background background = new Background(image); //图像背景对象
100    background.setApplicationBounds(bounds); //设置作用范围边界
101    root.addChild(background);
102    AmbientLight light = new AmbientLight(true,
103        new Color3f(Color.red)); //添加红色环境光源
104    light.setInfluencingBounds(bounds); //设置作用范围边界
105    root.addChild(light);
106    PointLight ptlight = new PointLight(new Color3f(Color.green),
107        new Point3f(3f, 3f, 3f), new Point3f(1f, 0f, 0f)); //添加绿色点光源
108    ptlight.setInfluencingBounds(bounds);
109    root.addChild(ptlight);
110    PointLight ptlight2 = new PointLight(new Color3f(Color.orange),
111        new Point3f(-2f, 2f, 2f), new Point3f(1f, 0f, 0f)); //添加橙色点光源
112    ptlight2.setInfluencingBounds(bounds);
113    root.addChild(ptlight2);
114    return root;
115 }
116 //捕捉函数
117 public BufferedImage capture() {
118     // 绘制off-screen图像
119     Dimension dim = cv.getSize();
120     view.stopView(); //停止视图
121     view.addCanvas3D(offScreenCanvas); //把off-screen画布对象附属到视图对象上
122     BufferedImage bImage =
123         new BufferedImage(dim.width, dim.height,
124             BufferedImage.TYPE_INT_RGB);
125     ImageComponent2D buffer =
126         new ImageComponent2D(ImageComponent.FORMAT_RGB, bImage);
127     offScreenCanvas.setOffScreenBuffer(buffer); //设置off-screen画布对象的缓存
128     view.startView(); //重新恢复视图
129     offScreenCanvas.renderOffScreenBuffer(); //开始off-screen绘制
130     offScreenCanvas.waitForOffScreenRendering(); //等待off-screen绘制的完成
131     bImage = offScreenCanvas.getOffScreenBuffer().getImage(); //获取绘制的图像
132     view.removeCanvas3D(offScreenCanvas); //与视图对象节点分离
133     return bImage;
134 }
135 //保存图像的方法
136 public void save(BufferedImage bImage) {
137     //把图像保存为文件
138     JFileChooser chooser = new JFileChooser(); //文件保存对话框
139     chooser.setCurrentDirectory(new File("."));
140     if (chooser.showSaveDialog(null) ==
141         JFileChooser.APPROVE_OPTION) {
142         File oFile = chooser.getSelectedFile();
143         try {

```



```
144      ImageIO.write(bImage, "jpeg", oFile); //保存图像文件
145  } catch (IOException ex) {
146      ex.printStackTrace();
147  }
148  }
149  }
150 }
```

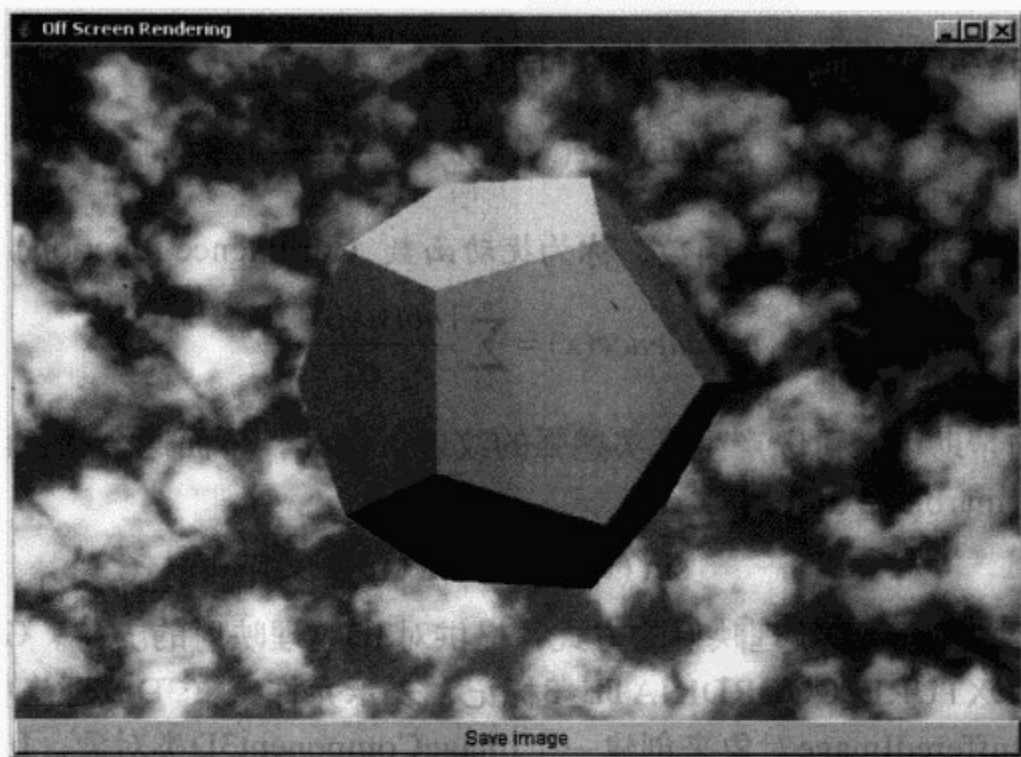


图12-12 通过离屏绘制创建图像

408

为了从场景的绘制结果捕捉图像，需要创建一个进行离屏绘制的Canvas3D画布对象，该离屏绘制画布对象直到捕捉图像结束之后，才被附属到视图对象节点上。

这个场景中包含有一个受到三个光源光照的十二面体，场景的背景是一幅天空图像，一个旋转插值器持续地对十二面体进行旋转。

capture方法（第117行）对图像进行实际的捕捉。首先，它调用stopView方法停止视图，然后把离屏画布对象附属到视图对象上，并把一个ImageComponent2D对象设置为此画布对象的离屏缓存对象。接下来调用startView方法，重新恢复视图。对renderOffScreenBuffer方法的调用，将启动离屏绘制过程，waitForOffScreenRendering方法等待离屏绘制的完成。最后，从画布对象获取绘制的图像，并把离屏画布对象与视图对象节点分离。

12.8 3D纹理

2D纹理技术把2D图像映射到3D物体的表面。3D纹理映射（或实体纹理映射）使用的则是3D立体纹理源。当然，3D纹理映射在产生和存储3D纹理数据方面要消耗更多的计算资源。尽管如此，3D纹理的优势在于它能用相关的简单映射函数，产生带有真实3D特征的真实感纹理特征。

典型情况下，3D纹理映射所使用的实体纹理，都是由计算机产生的合成纹理。通常3D纹理映射可以用于模拟大理石和木料等材质的天然纹理，这种方法也称为过程纹理（procedural texturing）。

为了生成和天然材质相似的纹理，需要在纹理模式中引入随机性或噪音。不过，在另一方面，完全随机的点并不能生成有用的纹理。所以，为随机点施加一定的平滑性和相关性处理是

很有必要的。

Perlin相关噪音 (Perlin's coherent noise) 函数是一种能够产生很好的视觉效果随机函数。noise(x)是一个光滑但又有一定随机性的函数，它在任意网格点（每一维的坐标都是整数坐标的点）上都生成一个称为梯度 (gradient) 的随机单位向量。对空间中任意一点x，围绕在点x周围的网格点都会对noise(x)的值产生影响。网格点g产生的贡献等于差向量x-g和g坐标处的随机单位向量的点积。对点x周围所有网格点g的贡献进行插值，就得到了noise(x)的最终值。

基于此噪音函数，可以定义一个分数的和式：

$$\text{perlinNoise}(x) = \sum_{i=1}^n \frac{\text{noise}(\beta^i x)}{\alpha^i}$$

另一个基于此噪音函数构建的函数，称为扰动函数 (turbulence function)：

$$\text{turbulence}(x) = \sum_{i=1}^n \frac{|\text{noise}(\beta^i x)|}{\alpha^i}$$

在噪音函数的帮助下，可以产生真实感强的纹理。例如，下面这个颜色函数能够产生很漂亮的大理石纹理 (marble texture)：

$$\sin(x + \text{turbulence}(p))$$

Java 3D以一种类似于2D纹理映射的方式，提供对3D纹理映射的支持。GeometryArray类对象可以包含一个TEXTURE_COORDINATE_3标记，表示允许定义3D纹理坐标。我们用一组用做3D纹理图像的BufferedImage对象来创建一个ImageComponent3D类对象。ImageComponent3D类对象，由一个Texture3D类对象进行引用。这里的Texture3D类对象，则是由一个Apperance类对象进行引用的节点组件。

程序清单12-10给出了3D纹理映射和Perlin噪音的应用情况，这个程序显示了一个带有3D大理石纹理的四面体（如图12-13所示）。所用的纹理是3D的，而且从四面体的不同侧面进行观察，纹理都表现出空间的连续性，该3D纹理是用Perlin噪音函数合成的。程序清单12-11定义了用于实体纹理映射的类。

409

程序清单12-10 PerlinNoise.java

```
1 package chapter12;
2
3 import javax.vecmath.*;
4 定义PerlinNoise类，表示Perlin噪声
5 public class PerlinNoise {
6     final static int B = 0x100;
7     int[] p = new int[2*B+2];
8     Vector3d[] g3 = new Vector3d[2*B+2];
9     //默认构造函数
10    public PerlinNoise() {
11        for (int i = 0; i < B; i++) {
12            p[i] = i;
13            double x = 2.0*Math.random() - 1.0;
14            double y = 2.0*Math.random() - 1.0;
15            double z = 2.0*Math.random() - 1.0;
16            g3[i] = new Vector3d(x, y, z);
17            g3[i].normalize();
```



```
18     }
19     for (int i = 0; i < B; i++) {
20         int k = p[i];
21         int j = (int)(Math.random()*B);
22         if (j >= B) j = B-1;
23         p[i] = p[j];
24         p[j] = k;
25     }
26     for (int i = 0; i < B+2; i++) {
27         p[B+i] = p[i];
28         g3[B+i] = g3[i];
29     }
30 }
31 //计算随机噪音值
32 public double noise(Point3d point) {
33     int bx0, bx1, by0, by1, bz0, bz1, b00, b10, b01, b11;
34     double rx0, rx1, ry0, ry1, rz0, rz1, sy, sz, a, b, c, d, t, u, v;
35     Vector3d q = null;
36     int i, j;
37
38     t = point.x + 0x1000;
39     bx0 = ((int)t) & 0xff;
40     bx1 = (bx0+1) & 0xff;
41     rx0 = t-(int)t;
42     rx1 = rx0-1;
43
44     t = point.y + 0x1000;
45     by0 = ((int)t) & 0xff;
46     by1 = (by0+1) & 0xff;
47     ry0 = t-(int)t;
48     ry1 = ry0-1;
49
50     t = point.z + 0x1000;
51     bz0 = ((int)t) & 0xff;
52     bz1 = (bz0+1) & 0xff;
53     rz0 = t-(int)t;
54     rz1 = rz0-1;
55
56     i = p[bx0];
57     j = p[bx1];
58
59     b00 = p[i+by0];
60     b10 = p[j+by0];
61     b01 = p[i+by1];
62     b11 = p[j+by1];
63
64     t = rx0*rx0*(3-2*rx0);
65     sy = ry0*ry0*(3-2*ry0);
66     sz = rz0*rz0*(3-2*rz0);
67
68     q = g3[b00+bz0]; u = rx0*q.x + ry0*q.y + rz0*q.z;
69     q = g3[b10+bz0]; v = rx1*q.x + ry0*q.y + rz0*q.z;
70     a = u + t*(v-u);
71 }
```

```

72     q = g3[b01+bz0]; u = rx0*q.x + ry1*q.y + rz0*q.z;
73     q = g3[b11+bz0]; v = rx1*q.x + ry1*q.y + rz0*q.z;
74     b = u + t*(v-u);
75
76     c = a + sy*(b-a);
77
78     q = g3[b00+bz1]; u = rx0*q.x + ry0*q.y + rz1*q.z;
79     q = g3[b10+bz1]; v = rx1*q.x + ry0*q.y + rz1*q.z;
80     a = u + t*(v-u);
81
82     q = g3[b01+bz1]; u = rx0*q.x + ry1*q.y + rz1*q.z;
83     q = g3[b11+bz1]; v = rx1*q.x + ry1*q.y + rz1*q.z;
84     b = u + t*(v-u);
85
86     d = a + sy*(b-a);
87
88     return c+sz*(d-c);
89 }
90
91 public double perlinNoise(Point3d pt,
92     double alpha, double beta, int n) {
93     double val;
94     double sum = 0;
95     double scale = 1;
96
97     for (int i = 0; i < n; i++) {
98         val = noise(pt);
99         sum += val / scale;
100         scale *= alpha;
101         pt.scale(beta);
102     }
103     return sum;
104 }
105 //计算turbulence函数值
106 public double turbulence(Point3d pt, double alpha,
107     double beta, int n) {
108     double val;
109     double sum = 0;
110     double scale = 1;
111
112     for (int i = 0; i < n; i++) {
113         val = noise(pt);
114         sum += Math.abs(val) / scale;
115         scale *= alpha;
116         pt.scale(beta);
117     }
118     return sum;
119 }
120 }

```

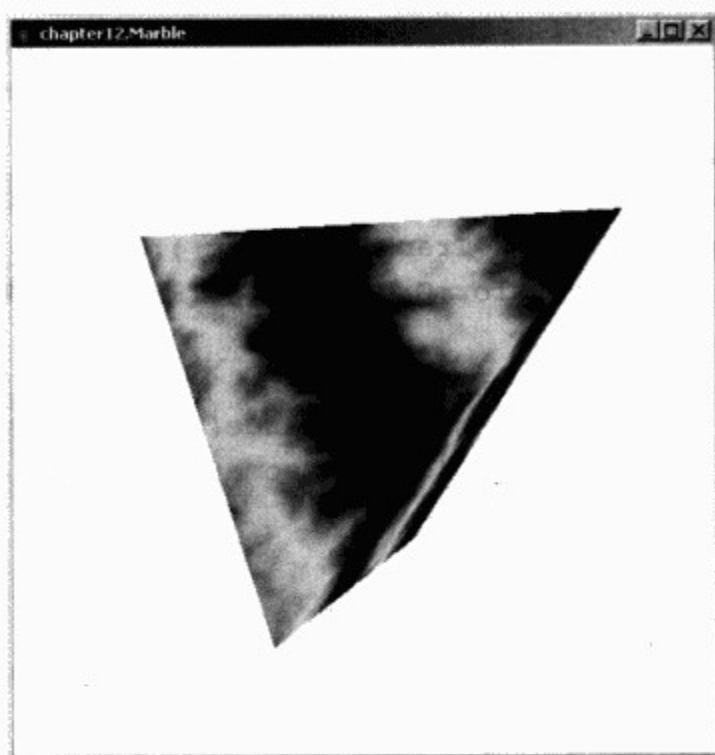



图12-13 计算机生成的3D大理石纹理

程序清单12-11 Marble.java

```
1 package chapter12;
2
3 import javax.vecmath.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.media.j3d.*;
7 import com.sun.j3d.utils.universe.*;
8 import com.sun.j3d.utils.geometry.*;
9 import com.sun.j3d.utils.image.*;
10 import javax.imageio.*;
11 import java.awt.image.*;
12 import java.io.*;
13 import java.applet.*;
14 import com.sun.j3d.utils.applet.MainFrame;
15 //定义Marble类, 继承自Applet类, 显示为一个旋转的带大理石纹理的正四面体
16 public class Marble extends Applet {
17     public static void main(String[] args) {
18         new MainFrame(new Marble(), 480, 480); //创建主窗口并设置大小
19     }
20
21     PerlinNoise pnoise = new PerlinNoise();
22
23     public void init() {
24         //创建Canvas3D画布对象
25         GraphicsConfiguration gc =
26             SimpleUniverse.getPreferredConfiguration();
27         Canvas3D cv = new Canvas3D(gc); //创建Canvas3D画布对象
28         setLayout(new BorderLayout()); //设置布局管理器
29         add(cv, BorderLayout.CENTER);
30         BranchGroup bg = createSceneGraph(); //创建场景图分支
31         bg.compile();
32         SimpleUniverse su = new SimpleUniverse(cv); //创建并设置SimpleUniverse对象
33         su.getViewingPlatform().setNominalViewingTransform();
```

```

34     su.addBranchGraph(bg);
35 }
36 //生成BranchGroup对象的私有函数, 创建场景图分支
37 private BranchGroup createSceneGraph() {
38     BranchGroup root = new BranchGroup();//场景图分支根节点
39     TransformGroup spin = new TransformGroup();
40     spin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
41     root.addChild(spin);
42     //构造图形对象
43     Appearance ap = createTextureAppearance();//外观对象
44     Shape3D shape = new Shape3D(createGeometry(), ap);//Shape3D叶节点
45     spin.addChild(shape);
46     //旋转
47     Alpha alpha = new Alpha(-1, 8000);
48     RotationInterpolator rotator =
49     new RotationInterpolator(alpha, spin);
50     BoundingSphere bounds = new BoundingSphere();//球体作用范围边界对象
51     rotator.setSchedulingBounds(bounds);
52     spin.addChild(rotator);
53     Background background = new Background(1.0f, 1.0f, 1.0f);//背景对象
54     background.setApplicationBounds(bounds);//设置作用范围边界
55     root.addChild(background);
56     return root;
57 }
58 //构造几何数据
59 GeometryArray createGeometry() {
60     IndexedTriangleArray ga = new IndexedTriangleArray(4,
61     TriangleArray.COORDINATES | TriangleArray.NORMALS |
62     TriangleArray.TEXTURE_COORDINATE_3, 12); //创建带索引三角数组对象
63     ga.setCoordinate(0, new Point3f(0.5f, 0.5f, 0.5f)); //设定顶点值
64     ga.setCoordinate(1, new Point3f(0.5f, -0.5f, -0.5f));
65     ga.setCoordinate(2, new Point3f(-0.5f, 0.5f, -0.5f));
66     ga.setCoordinate(3, new Point3f(-0.5f, -0.5f, 0.5f));
67     int[] coords = {0, 1, 2, 0, 3, 1, 1, 3, 2, 2, 3, 0};
68     ga.setNormal(0, new Vector3f(1f, 1f, -1f)); //设置各顶点处的法向量
69     ga.setNormal(1, new Vector3f(1f, -1f, 1f));
70     ga.setNormal(2, new Vector3f(-1f, -1f, -1f));
71     ga.setNormal(3, new Vector3f(-1f, 1f, 1f));
72     int[] norms = {0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3};
73     ga.setCoordinateIndices(0, coords); //设置顶点索引数组
74     ga.setNormalIndices(0, norms); //设定顶点法向量索引数组
75     ga.setTextureCoordinate(0, 0, new TexCoord3f(1f, 1f, 1f)); //设置各顶点纹理坐标
76     ga.setTextureCoordinate(0, 1, new TexCoord3f(1f, 0f, 0f));
77     ga.setTextureCoordinate(0, 2, new TexCoord3f(0f, 1f, 0f));
78     ga.setTextureCoordinate(0, 3, new TexCoord3f(0f, 0f, 1f));
79     ga.setTextureCoordinateIndices(0, 0, coords); //设定纹理坐标索引数组
80     return ga;
81 }
82 //创建纹理外观对象
83 Appearance createTextureAppearance(){
84     Appearance ap = new Appearance();//外观对象
85     BufferedImage[] img = new BufferedImage[128]; //128幅图像的数组
86     for (int i = 0; i < 128; i++) {
87         img[i] = new BufferedImage(128, 128,

```



```

88     BufferedImage.TYPE_INT_ARGB);
89     for (int r = 0; r < 128; r++) {
90         for (int c = 0; c < 128; c++) { //调用turbulence方法计算像素值
91             double v = pnoise.turbulence
92                 (new Point3d(c/32.0, r/32.0, i/32.0), 2, 2, 8);
93             int rgb = (int)((0.55+0.35*Math.sin(3*(c/32.0+v)))*256);
94             rgb = ((rgb << 16) | (rgb << 8) | rgb);
95             img[i].setRGB(c, r, rgb);
96         }
97     }
98 }
99 ImageComponent3D image = //3D图像对象
100     new ImageComponent3D(ImageComponent3D.FORMAT_RGB, img);
101 Texture3D texture = //3D纹理对象
102     new Texture3D(Texture.BASE_LEVEL, Texture.RGBA,
103     image.getWidth(), image.getHeight(), image.getDepth());
104 texture.setImage(0, image);
105 texture.setEnabled(true);
106 texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);
107 texture.setMinFilter(Texture.BASE_LEVEL_LINEAR); //设置插值滤波方式
108 ap.setTexture(texture);
109 return ap;
110 }
111 }

```

PerlinNoise类基于Perlin的原始方法，实现了噪音和扰动函数，该类以一个 $256 \times 256 \times 256$ 大小的网格点集作为初始的随机向量。为了减少存储开销，只生成一个用于存储随机梯度值的一维数组g3和排列p（第7~8行）。网格点 (i, j, k) 处的梯度，根据公式 $g(i, j, k) = g3(k + p(j + p(i)))$ 进行计算。

对网格点的贡献进行插值计算，就可以得到noise函数的值（第32行）。

perlinNoise方法（第91行）计算了噪音函数的分数和，turbulence方法（第106行）计算的是扰动函数的值。

Marble（大理石）类显示一个带实体大理石纹理的旋转四面体，其场景图如图12-14所示。

createGeometry方法创建了一个四面体对象，并以TEXTURE_COORDINATE_3标记指定其3D纹理的坐标。基于各个顶点的空间坐标计算它们的纹理坐标，并归一化到 $[0, 1]$ 的范围内。

createTextureAppearance方法（第83行）创建四面体的外观对象，并且将Appearance对象设置为引用一个Texture3D对象。同时，还创建了一个包含128个BufferedImage类对象的数组，其中每个数组元素都是一幅基于扰动函数生成的大理石纹理图像，图像的大小为 128×128 （第85~96行）。该BufferedImage数组形成了“纹理立方体”，程序中用它来创建了一个ImageComponent3D类对象，该ImageComponent3D对象随后传递给作为纹理映射源的Texture3D对象。

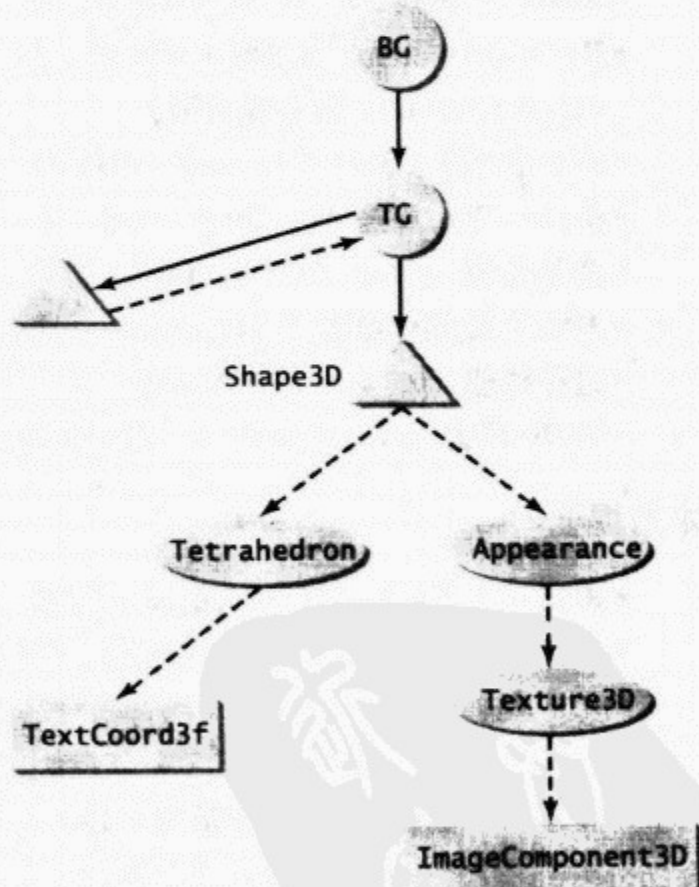


图12-14 大理石四面体的场景图分支结构

主要的类和方法

- `javax.media.j3d.Point3d.interpolate(...)` 执行线性插值的方法。
- `javax.media.j3d.GeometryUpdater` 用以修改活动场景中的几何数据的接口。
- `javax.media.j3d.GeometryArray.updateData(GeometryUpdater)` 通过作为参数的`GeometryUpdater`对象对几何数据进行更新的方法。
- `javax.media.j3d.Sound` 声音对象的基类。
- `javax.media.j3d.BackgroundSound` 封装背景声音对象的类。
- `javax.media.j3d.PointSound` 封装3D点声源对象的类。
- `javax.media.j3d.ConeSound` 封装发散角度有限的3D点声源对象的类。
- `javax.media.j3d.MediaContainer` 一种作为声音数据容器的节点组件类。
- `com.sun.j3d.utils.universe.Viewer.createAudioDevice(...)` 初始化默认音频设备的方法。
- `javax.media.j3d.Canvas3D.setOffScreenBuffer(...)` 设置离屏绘制缓存的方法。
- `javax.media.j3d.Canvas3D.renderOffScreenBuffer(...)` 启动离屏缓存绘制的方法。
- `javax.media.j3d.Canvas3D.waitForOffScreenRendering(...)` 等待离屏绘制完成的方法。
- `javax.media.j3d.Canvas3D.getOffScreenBuffer(...)` 获取离屏绘制缓存的方法。
- `javax.media.j3d.Texture3D` 用于3D纹理的节点组件类。
- `javax.vecmath.TexCoord3f` 表示3D纹理坐标的类。
- `javax.media.j3d.ImageComponent3D` 用于定义3D图像组件的节点组件类。

关键术语

- **3DBézier曲线** (3D Bézier curves) 由一组控制点定义的多项式参数曲线。
- **Bernstein多项式或Bernstein基** (Bernstein polynomial or Bernstein basis) 一种用于逼近 (approximation) 的特殊多项式。
- **Bézier曲面** (Bézier surface) 由控制点网格定义的多项式参数曲面。
- **deCasteljau算法** (deCasteljau algorithm) 通过一系列线性插值来计算Bézier曲线上的点的算法。
- **离屏绘制** (offscreen rendering) 把场景绘制到内存中的缓存而不是计算机屏幕上的过程。
- **3D纹理或实体纹理** (3D texture or solid texture) 定义为3D立体的纹理。
- **过程纹理** (procedural texturing) 以计算的方式创建合成纹理的方法。
- **Perlin噪音** (Perlin's noise) 用于生成合成纹理的相关随机函数。

本章提要

- 本章讨论了与3D图形相关的若干问题。
- 曲线和曲面是针对可视对象的几何属性进行建模的重要工具。本章讨论了Bézier曲线和曲面的实现，其基本工具为deCasteljau算法，该算法以线性插值法对曲线上的点求值，并可用于分割曲线。
- Java 3D通过`Sound`类叶节点及其子类提供对声音的支持。向场景中添加声音与添加其他节点是类似的，能在场景中包含声音的好处是，可以支持音频与可视对象的自然连接。
- 本章讨论了一种用于生成简单阴影的方法，我们用原物体投影得到的平面多边形对象来模拟阴影。
- 通过`BY_REFERENCE`模式和`GeometryUpdater`接口，可以对几何属性进行动态修改。
- `Canvas3D`类支持离屏绘制模式，可以用来捕捉并保存所绘制的图像。
- 3D纹理映射把3D立体图像映射到可视对象上。Java 3D中，支持3D纹理映射的框架与2D纹理映射的框架相同。通常3D纹理数据用某种受控的随机过程进行合成，Perlin噪音函数与扰动函数，可以用来生成类似大理石、木料或其他自然纹理的模式。

复习题

- 12.1 一条Bézier曲线由以下控制点定义： $p_0(0, 0, 0)$, $p_1(1, 0, 0)$, $p_2(1, 1, 0)$, $p_3(1, 1, 1)$ ；请用deCasteljau算法计算曲线上如下位置的点：(a) $t = 1/2$ ；(b) $t = 1/3$ ；(c) $t = 2/3$ 。
- 12.2 请在以下位置用deCasteljau算法，将习题12.1中定义的Bézier曲线分割成两段：
(a) $t = 1/2$ ；(b) $t = 1/3$ ；(c) $t = 2/3$ 。
- 12.3 试推导求三次Bézier曲面上任一点的法向量的公式。
- 12.4 设在空间中的点 $(0, 0, 1)$ 位置放置有一个点光源，并在xy平面上投下阴影。请推导从任一点投影到此平面的变换矩阵。

416

编程练习

- 12.1 类似于第4章中定义的二次曲线的情况，请通过转换为一系列Bézier曲线，实现一条3D B样条曲线。
- 12.2 请编写一个三次Bézier曲面编辑器程序。该程序把曲面的各控制点显示为正方形点，并允许用户通过鼠标操作选择与移动某个控制点，并根据用户的操作对曲面进行更新。
- 12.3 请编写一个Java 3D程序，显示一个Bézier曲面及其在一个平面上投下的阴影。
- 12.4 请编写一个类似于程序清单12-7的Java 3D程序，显示一个球体及其在一面墙上投下的阴影。请添加一个OrbitBehavior对象，以根据鼠标动作控制视图平台的变换。
- 12.5 请编写一个Java 3D程序，显示一个有100个点的PointArray对象，请用行为来随机地独立移动该几何对象中的点。
- 12.6 请编写一个Java 3D程序来显示一个旋转的圆锥体，创建一个与该圆锥体关联的ConeSound对象，使该ConeSound对象的方向与可视圆锥体的方向一致。
- 12.7 请编写一个Java 3D程序，显示两个面板和一个按钮。第一个面板显示一个旋转的文字字符串，点击按钮时，捕获该3D场景的图像，并在第二个面板上显示所捕获的静止图像。
- 12.8 请编写一个程序，画出Perlin噪音的x分量。
- 12.9 定义木材纹理为：

$$g(p) = \text{turbulence}(p) * 20$$

$$\text{grain}(p) = g(p) - \lfloor g(p) \rfloor$$

请编写一个Java 3D程序，显示一个带实体木材纹理的旋转立方体。

417

附录A 计算机图形学的数学背景

A.1 引言

本附录简要回顾了若干与计算机图形学密切相关的数学主题。计算机图形学很大程度上依靠数学工具来精确地对图形对象及图形系统进行建模、变换和处理。

几何学尤其是解析几何，为图形对象的建模提供了一种有效的工具。解析几何中的坐标系使几何模型的数值表示变得很方便。在计算机图形系统中，这种建模方案是很基础的，因为几何实体、属性及变换与计算机图形学的核心问题直接相关。

复数和四元数是具有高度的结构特性的代数系统，它们具有直接的几何解释，这使得它们成为解决几何和图形学中某些问题的有用工具。

线性代数这门代数学科将实数、复数和四元数系统扩展到任意维的向量空间，系统地研究了向量的属性和关系。矩阵为线性变换和向量提供了具体的表示，它们可以直接在计算机中实现，从而表示和计算机图形学有关的几何变换。

微积分学可以用于解决求切线和曲面法线这样的问题，这些问题与图形学中的一些建模及绘制问题有关。

图论研究了一种称为图（graph）的离散数学结构，它在计算机科学和图形学中有着广泛的应用。

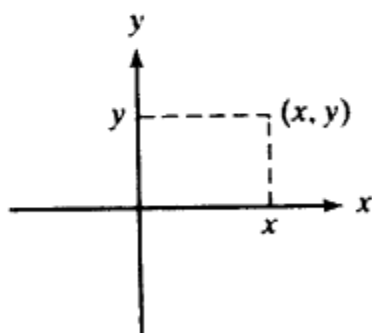
仅仅一章或一个附录的篇幅，不可能全面覆盖这些数学问题，在这里，我们介绍与计算机图形学相关的最基本的概念和结论，这些材料可以作为本书中相关数学问题的快速参考。

A.2 解析几何

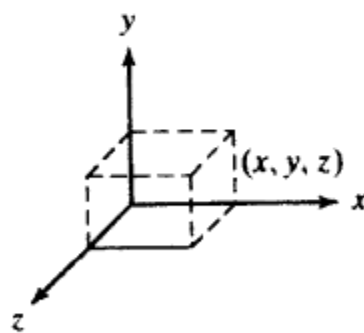
A.2.1 坐标系

一个几何对象以代数方式在坐标系中进行表示。如图A-1所示，平面上的一个点由2D笛卡儿坐标系中的 x 坐标和 y 坐标表示。

类似地，如图A-2所示，利用3D笛卡儿坐标系，3D空间的一个点可以用三个实数所组成的一个元组 (x, y, z) 来表示。



图A-1 有 x 轴和 y 轴的2D坐标系



图A-2 有 x 轴、 y 轴和 z 轴的3D坐标系

利用所建立的点和坐标之间的一一对应关系，几何属性可以表示为代数量。例如，平面上两个点 (x_1, y_1) 和 (x_2, y_2) 之间的距离，可以由以下距离公式（distance formula）给出：

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

3D空间的点 (x_1, y_1, z_1) 和点 (x_2, y_2, z_2) 之间的距离可以由以下距离公式给出:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

A.2.2 2D方程

几何对象通常可以表示为代数方程。一个几何对象上的点具有这样的特征: 它们的坐标都满足相应的方程式。在2D空间中, 方程 $F(x, y) = 0$ 或显式函数 $y = f(x)$ 定义了一条曲线。

一条直线可以由一般的线性方程进行表征:

$$Ax + By + C = 0$$

点斜式 (point-slope equation) 是一种更为方便的线性方程。如图A-3所示, 给定直线上的两点 (x_0, y_0) 和 (x_1, y_1) , x 和 y 方向上的差分分别定义为: $\Delta x = x_1 - x_0$, $\Delta y = y_1 - y_0$ 。直线的斜率定义为比值 $k = \Delta y / \Delta x$, 这个比值对于任何给定的直线都是固定的常量。给出了直线上的一点 (x_0, y_0) 和斜率 k 之后, 直线的方程就可以按下面的点斜式写出:

$$y - y_0 = k(x - x_0)$$

以点 (x_0, y_0) 为圆心、 R 为半径的圆可以表示为如下方程:

$$(x - x_0)^2 + (y - y_0)^2 = R^2$$

称为圆锥曲线 (conic section) 的一族曲线, 可以由一般二次方程表示为:

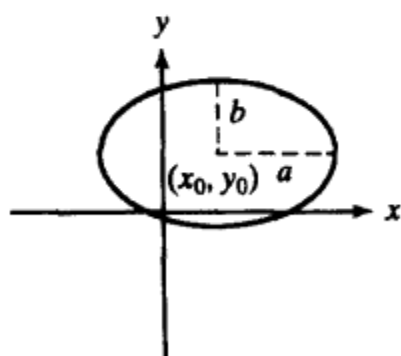
$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$$

圆锥曲线可以分为三个主要的类型: 椭圆、抛物线和双曲线。下面给出了它们的标准方程 (参见图A-4、A-5、A-6)。

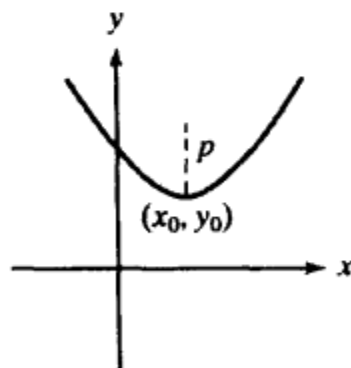
$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1 \quad (\text{椭圆})$$

$$(x - x_0)^2 = 4p(y - y_0) \quad (\text{抛物线})$$

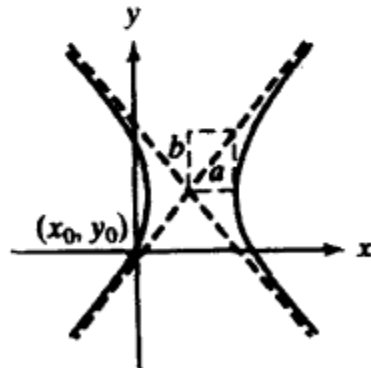
$$\frac{(x - x_0)^2}{a^2} - \frac{(y - y_0)^2}{b^2} = 1 \quad (\text{双曲线})$$



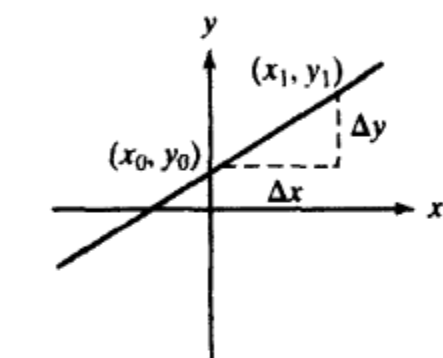
图A-4 椭圆可以用二次方程表示



图A-5 抛物线可以用二次方程表示



图A-6 双曲线可以用二次方程表示



图A-3 直线可由线性方程表示

420

421

A.2.3 参数方程

曲线的参数方程使用第三个变量 t 作为一个独立的变量,把 x 坐标和 y 坐标定义成 t 的函数。例如, (a, b) 方向的经过点 (x_0, y_0) 的直线有参数方程 (见图A-7):

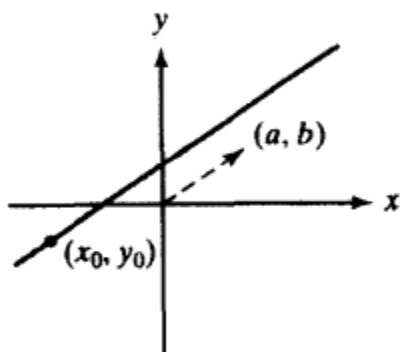
$$x = x_0 + at$$

$$y = y_0 + bt$$

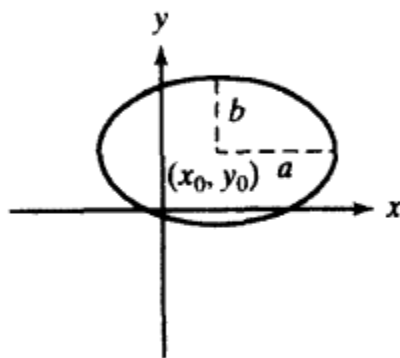
如图A-8所示,椭圆可以表达成参数方程:

$$x = x_0 + a \cos t$$

$$y = y_0 + b \sin t$$



图A-7 直线可以由参数方程表示



图A-8 椭圆可以由参数方程表示

422

A.2.4 3D方程

在3D空间中,方程 $F(x, y, z) = 0$ 或显式函数 $z = f(x, y)$ 表示空间中的曲面。

一般的线性方程表示一个平面

$$Ax + By + Cz + D = 0$$

平面的法线是垂直于平面的一个方向,如果平面的法线由 (a, b, c) 给出, (x_0, y_0, z_0) 为平面上一个定点,那么平面的方程可以写成:

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$$

以 (x_0, y_0, z_0) 为球心、 R 为半径的球 (sphere), 有如下标准方程:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = R^2$$

3D空间中的直线可以表示成两个平面的相交,或者表示成参数方程:

$$x = x_0 + at$$

$$y = y_0 + bt$$

$$z = z_0 + ct$$

曲面的参数方程涉及两个变量:

$$x = f(u, v)$$

$$y = g(u, v)$$

$$z = h(u, v)$$

例如,经过 (x_0, y_0, z_0) , (x_1, y_1, z_1) , (x_2, y_2, z_2) 三个点的平面,有如下的参数方程:

$$x = x_0 + (x_1 - x_0)u + (x_2 - x_0)v$$

$$y = y_0 + (y_1 - y_0)u + (y_2 - y_0)v$$

$$z = z_0 + (z_1 - z_0)u + (z_2 - z_0)v$$

A.3 复数和四元数

A.3.1 复数

复数可以用来表示2D几何对象和变换。复数集是实数系统的扩展，一个复数可以写成：

$$a + bi$$

其中， a, b 是实数，且 $i^2 = -1$ 。在复数上可以定义加法和乘法运算，令两个复数 $z = a + bi, w = c + di$ ，则这两种运算可以定义为：

$$z + w = (a + c) + (b + d)i$$

$$z \cdot w = (ac - bd) + (ad + bc)i$$

423

复数的乘法运算可以根据分配律和 $i^2 = -1$ 得到。例如

$$(2 - i) + (3 + 4i) = 5 + 3i$$

$$(2 - i)(3 + 4i) = 6 + 8i - 3i - 4i^2 = 10 + 5i$$

带有乘法和加法运算的复数集具有某些和实数集相似的性质，这两种运算都是可结合与可交换的。乘法对于加法满足分配律，它们都有幺元（0和1）和逆元，加法的逆运算是减法，而乘法的逆是除法。因此，和实数系统一样，复数集加上这两个运算，便形成了一个代数结构，这种代数结构称为域（field）。像加法一样，两个复数的减法分别对实部和虚部进行。例如，

$$(2 - i) - (3 + 4i) = -1 - 5i$$

除法运算要更复杂一点，在介绍它之前，还必须引入几个新的术语。 $z = a + bi$ 的共轭复数（complex conjugate）和绝对值定义为：

$$\bar{z} = a - bi$$

$$|z| = \sqrt{a^2 + b^2}$$

复数的绝对值是一个非负的实数，易验证得 $z\bar{z} = |z|^2$ 。两个复数 w 和 z 的除法，可以通过分母和分子同乘以分母的共轭复数来计算，也就是：

$$\frac{w}{z} = \frac{w\bar{z}}{z\bar{z}} = \frac{w\bar{z}}{|z|^2}$$

例如，

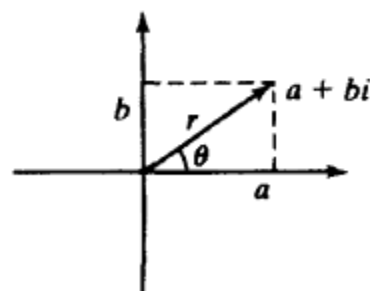
$$\frac{2 - i}{3 + 4i} = \frac{(2 - i)(3 - 4i)}{(3 + 4i)(3 - 4i)} = \frac{2 - 11i}{25} = (2/25) - (11/25)i$$

上面介绍的标准的复数直角坐标形式，在某些情况下并不是最方便的表示方法，复数可以写成极坐标形式（polar form）

$$z = r(\cos \theta + i \sin \theta) = re^{i\theta}$$

其中， $r = |z|$ ，而 θ 则是一个称为辐角（argument）的实数。等式 $e^{i\theta} = \cos \theta + i \sin \theta$ 就是著名的欧拉公式（Euler's identity），它建立了三角函数和指数函数之间的关系。利用极坐标形式来计算复数的乘法、除法、求幂及求根，会显得更为简单。

复数的几何解释就是平面上的点，复数 $z = a + bi$ 和2D点 (a, b) 或从坐标原点到 (a, b) 点的向量相关联（见图A-9）。绝对值 r 相应于指定点到原点的距离，或者说是对应向量的长度。角度 θ 是对应向量和x轴之间的夹角，两个复数的相加



图A-9 复数可以用于表示几何对象

424

对应于向量的相加，两个复数相乘的几何解释可以从它的极坐标形式看出：

$$z_1 z_2 = r_1 e^{i\theta_1} r_2 e^{i\theta_2} = (r_1 r_2) e^{i(\theta_1 + \theta_2)}$$

因此，所形成的向量的长度就是两个向量长度的积，新向量的角度就是两个向量角度的和。程序清单A-1显示了一个实现复数运算的Java类。

程序清单A-1 Complex.java

```

1 public class Complex { //定义复数类Complex
2     public static void main(String[] args) {
3         Complex z = new Complex(1, -2); //构造一个复数对象，其值为1-2i
4         Complex w = new Complex(3, 4); //构造一个复数对象，其值为 3+4i
5         System.out.println("z = " + z);
6         System.out.println("w = " + w);
7         System.out.println("|z| = " + z.abs());
8         System.out.println("|w| = " + w.abs());
9         System.out.println("arg z = " + z.arg());
10        System.out.println("arg w = " + w.arg());
11        System.out.println("conj z = " + z.conj());
12        System.out.println("conj w = " + w.conj());
13        System.out.println("z + w = " + z.add(w));
14        System.out.println("z - w = " + z.sub(w));
15        System.out.println("z * w = " + z.mul(w));
16        System.out.println("z / w = " + z.div(w));
17    }
18
19    private double x = 0.0;
20    private double y = 0.0;
21
22    public Complex() {
23    }
24
25    public Complex(double x, double y) {
26        this.x = x;
27        this.y = y;
28    }
29    //返回实部的方法
30    public double getX() {
31        return x;
32    }
33    //返回虚部的方法
34    public double getY() {
35        return y;
36    }
37    //将实部设为x的方法
38    public void setX(double x) {
39        this.x = x;
40    }
41    //将虚部设为y的方法
42    public void setY(double y) {
43        this.y = y;
44    }

```



```
45 //求复数绝对值的方法
46 public double abs() {
47     return Math.sqrt(x*x + y*y);
48 }
49 //求复数辐角的方法
50 public double arg() {
51     return Math.atan2(y, x);
52 }
53 //求复数的共轭复数的方法
54 public Complex conj() {
55     return new Complex(x, -y);
56 }
57 //复数的加法
58 public Complex add(Complex other) {
59     double a = x + other.getX();
60     double b = y + other.getY();
61     return new Complex(a, b);
62 }
63 //复数的减法
64 public Complex sub(Complex other) {
65     double a = x - other.getX();
66     double b = y - other.getY();
67     return new Complex(a, b);
68 }
69 //复数的乘法
70 public Complex mul(Complex other) {
71     double a = x * other.getX() - y * other.getY();
72     double b = x * other.getY() + y * other.getX();
73     return new Complex(a, b);
74 }
75 //复数的除法
76 public Complex div(Complex other) {
77     double a = x * other.getX() + y * other.getY();
78     double b = -x * other.getY() + y * other.getX();
79     double d = other.abs();
80     d = d * d;
81     return new Complex(a/d, b/d);
82 }
83 //将复数转化为字符串的形式以便输出
84 public String toString() {
85     if (y >= 0)
86         return "" + x + "+" + y + "i";
87     else
88         return "" + x + "" + y + "i";
89 }
90 }
```

Complex类封装了复数的概念，它包含两个成员变量x和y，分别用来表示复数的实部和虚部（第19~20行），这两个成员变量都可以由getter方法和setter方法进行访问。

默认构造函数生成了复数0，其他构造函数按照指定的实部和虚部来生成复数（第25~28行）。

复数的绝对值和辐角可以分别由abs方法和arg方法获得。conj方法返回一个新的Complex对象，这个对象是当前对象的共轭复数。复数的加法、减法、乘法和除法分别由add、sub、mul、div方法实现，每个方法都将一个Complex参数作为第二操作数，并返回一个新的Complex对象

作为运算结果。

toString方法（第84行）经过重写而提供了一种复数的惯用字符串表示方法，表示成“3-4i”的形式。

426

main方法用于测试Complex类，它创建了两个复数，这两个复数对象通过隐式地调用toString方法而打印出来。同时，将这两个数的所有运算结果也打印出来了，程序的输出结果显示如下：

```
z = 1.0-2.0i
w = 3.0+4.0i
|z| = 2.23606797749979
|w| = 5.0
arg z = -1.1071487177940904
arg w = 0.9272952180016122
conj z = 1.0+2.0i
conj w = 3.0-4.0i
z + w = 4.0+2.0i
z - w = -2.0-6.0i
z * w = 11.0-2.0i
z / w = -0.2-0.4i
```

A.3.2 四元数

四元数制是复数系统的一个扩展。一个四元数可以表示成：

$$q = a + bi + cj + dk$$

其中 a, b, c, d 是实数，而 i, j, k 是满足以下规则的元素：

$$i^2 = j^2 = k^2 = ijk = -1$$

假设乘法满足结合律，同样可得：

$$ij = k = -ji \quad jk = i = -kj \quad ki = j = -ik$$

类似于复数的加减法，四元数的加法和减法定义为分量运算（componentwise operation）。与复数的乘法类似，四元数的乘法的定义也是根据乘法对加法满足分配律，而且 i, j, k 满足上述关系。

两个四元数 $q = a + bi + cj + dk$ 和 $p = e + fi + gj + hk$ 的加法定义为：

$$q + p = (a + e) + (b + f)i + (c + g)j + (d + h)k$$

四元数的乘法用分配律和 i, j, k 的性质定义：

$$\begin{aligned} q \cdot p = & (ae - bf - cg - dh) + (af + be + ch - dg)i \\ & + (ag - bh + ce + df)j + (ah + bg - cf + de)k \end{aligned}$$

例如，

$$\begin{aligned} & (1 - i + j - 2k) \cdot (2 + i - k) \\ = & 1 \cdot 2 + 1 \cdot i - 1 \cdot k - 2 \cdot i - i \cdot i + i \cdot k + 2 \cdot j + j \cdot i - j \cdot k - 4 \cdot k - 2k \cdot i + 2k \cdot k \\ = & 2 + i - k - 2i + 1 - j + 2j - k + i - 4k - 2j - 2 \\ = & 1 - j - 6k \end{aligned}$$

四元数乘法不是可交换的，所以一般来说， $p \cdot q$ 不等于 $q \cdot p$ 。但是，与实数和复数系统类似，其他的代数学性质，例如结合律、分配律、幺元和逆元的存在性，都是有效的。

四元数制被称为一个反称域 (skew field), 或者说是一个非交换域 (noncommutative field)。[427]
除了不一定满足交换律外, 反称域满足数域的所有性质。

四元数 $q = a + bi + cj + dk$ 的共轭和绝对值定义为,

$$\bar{q} = a - bi - cj - dk$$

$$|q| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

可以证明, $\overline{p \cdot q} = \bar{p} \cdot \bar{q}$ 且 $q \cdot \bar{q} = \bar{q} \cdot q = |q|^2$ 。后一个等式可用于按照定义复数除法的类似方式来定义四元数的除法, 这和复数除法的方法是一样的。例如,

$$\frac{-k}{1+2i-j} = \frac{-k(1-2i+j)}{(1+2i-j)(1-2i+j)} = \frac{-k+2j+i}{6} = (1/6)i + (1.3)j - (1/6)k$$

纯四元数是实数部分为0的四元数 (例如 $xi + yj + zk$)。一种对四元数进行几何解释的方法, 就是将3D空间中的一个点和一个纯四元数联系起来。

四元数在表示3D旋转时尤为有用, 相关细节问题将在A.5节中介绍。

A.4 线性代数

A.4.1 向量空间

F 域 (例如实数域或复数域) 上的一般向量空间是这样的一个集合 V , 在其上定义了一个加法 $+: V \times V \rightarrow V$ 和一个数乘 $\cdot: F \times V \rightarrow V$, 这些运算满足下列一些条件。

对于任给的 $u, v, w \in V$ 及 $k, l \in F$,

$$u + v = v + u$$

$$(u + v) + w = u + (v + w)$$

存在0向量, 使得 $u + 0 = u$

每个向量 u 都存在逆 $-u$, 使得 $u + (-u) = 0$:

$$k(u + v) = ku + kv$$

$$(k + l)u = ku + lu$$

$$(kl)u = k(lu)$$

$$1u = u$$

作为一个具体的例子, 一个典型的 n 维向量只是 n 个实数的有序元组。例如, $(3, 4)$, $(1, 0, -2)$, $(4, 5, -3, 1)$ 分别为2D、3D和4D向量。

所有这样的 n 维向量的集合, 形成了实数域上的 n 维向量空间, 向量空间的加法和乘法运算定义为分量运算 (componentwise operation)。两个向量的加法定义为:

$$(a_1, a_2, \dots, a_n) + (b_1, b_2, \dots, b_n) = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$$

数乘定义为:

$$k(a_1, a_2, \dots, a_n) = (ka_1, ka_2, \dots, ka_n)$$

这个特定的 n 维向量空间可表示为 R^n 。

复数集可以看做是实数域上的2D向量空间, 四元数是实数域上的4D向量空间, 或者说是复数域上的2D向量空间。

内积 (inner product) 也称为点积或标量积, 在 R^n 上定义为:

[428]

$$(a_1, a_2, \dots, a_n) \cdot (b_1, b_2, \dots, b_n) = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

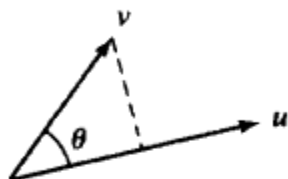
向量 v 的范数 (norm) 定义为: $\|v\| = \sqrt{v \cdot v}$ 。

利用直角坐标系, 2D几何平面上的一个点可以与一个2D向量相关联。这种关联建立起了几何平面和代数向量空间之间的一一对应关系。类似地, 3D几何空间和3D向量空间可以建立一一对应的关系。这样的联系和解析几何中所用的概念完全一致。

从几何意义上讲, 向量的范数就是向量的长度, 两个向量的内积可以表示成

$$v \cdot u = \|v\| \|u\| \cos \theta$$

其中, θ 是两个向量的夹角, 图A-10图解了内积的含义。如果 u 是一个单位向量, $\|u\| = 1$, 那么它们的内积可以解释为 v 在 u 方向上的投影。



图A-10 内积的几何解释

如果两个向量的夹角为 90° 或 $\pi/2$ 弧度, 那么称这两个向量是正交的 (orthogonal) 或垂直的 (perpendicular)。由内积的性质可知, 当 $v \cdot u = 0$ 时, v 和 u 是正交的。

A.4.2 线性变换和矩阵

从数学意义上说, 一个从空间 V 到空间 W 的变换 T (transformaion或transform) 是 V 到 W 的一个映射 (一个函数):

$$T: V \rightarrow W$$

对于 V 中每个点 v , 在 W 中都有与之相应的点 $T(v)$ 。如果映射 T 一一对应并且是映射成的, 那么它有逆映射:

$$T^{-1}: W \rightarrow V$$

在计算机图形中, 空间 V 和 W 通常是2D向量空间 R^2 或3D向量空间 R^3 , 整个变换族是数量庞大且很复杂的。在图形学中, 通常只考虑某些特殊的变换族。

如果对任给的向量 u, v 和标量 a, b , 都满足以下等式, 则变换 T 是线性的:

$$T(au + bv) = aT(u) + bT(v)$$

线性变换便于进行代数操作和计算机表示与处理。从 n 维向量空间 R^n 到 m 维向量空间 R^m 的一个普通线性变换, 可以用一个 $m \times n$ 矩阵表示。矩阵是数的矩形阵列, 一个 $m \times n$ 矩阵有如下表示形式:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

例如, 下面是一个 2×3 矩阵

$$\begin{bmatrix} 3 & -1 & 2 \\ 1 & 0 & -3 \end{bmatrix}$$

两个具有相同尺寸的矩阵的加法定义为逐项相加 (entrywise operation)。例如,

$$\begin{bmatrix} 3 & -1 & 2 \\ 1 & 0 & -3 \end{bmatrix} + \begin{bmatrix} 1 & 0 & -3 \\ -1 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 3+1 & -1+0 & 2-3 \\ 1-1 & 0+2 & -3+5 \end{bmatrix} = \begin{bmatrix} 4 & -1 & -1 \\ 0 & 2 & 2 \end{bmatrix}$$

对于矩阵 A 和 B , 当且仅当 A 的列数等于 B 的行数时, 可以定义矩阵乘法 AB 。乘积 AB 的每一

项都是由A的一行和B的一列的相应项相乘而后求和得到。例如，

$$\begin{aligned} & \begin{bmatrix} 3 & -1 & 2 \\ 1 & 0 & -3 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 2 \\ -3 & 5 \end{bmatrix} \\ &= \begin{bmatrix} 3 \times 1 + (-1) \times 0 + 2 \times (-3) & 3 \times (-1) + (-1) \times 2 + 2 \times 5 \\ 1 \times 1 + 0 \times 0 + (-3) \times (-3) & 1 \times (-1) + 0 \times 2 + (-3) \times 5 \end{bmatrix} \\ &= \begin{bmatrix} -3 & 5 \\ 10 & -16 \end{bmatrix} \end{aligned}$$

矩阵也能表示向量，一个 m 维向量可以用一个 $m \times 1$ 的列矩阵表示。例如，下面的矩阵表示一个3D向量：

$$\begin{bmatrix} -1 \\ 4 \\ 0 \end{bmatrix}$$

可以利用矩阵来建立线性变换。令A为一个 $m \times n$ 矩阵， v 为表示一个向量的 $n \times 1$ 的列矩阵，则从 R^n 到 R^m 的线性变换可定义为矩阵乘法：

$$u = Av$$

例如，下面的矩阵等式定义了一个从 R^3 到 R^2 的线性变换：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 \\ 1 & 0 & -3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

430

反过来，任何从 R^m 到 R^n 的线性变换则可以用一个合适的 $n \times m$ 矩阵进行矩阵表示。给定一个 $m \times n$ 矩阵A，其转置 A^T 定义为如下的 $n \times m$ 矩阵：

$$A^T = \begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{m,n} \end{bmatrix}$$

例如，

$$\begin{bmatrix} 3 & -1 & 2 \\ 1 & 0 & -3 \end{bmatrix}^T = \begin{bmatrix} 3 & 1 \\ -1 & 0 \\ 2 & -3 \end{bmatrix}$$

$n \times n$ 矩阵称为方阵。一个方阵表示一个从向量空间 R^n 到 R^n 的线性变换。单位矩阵I是有如下形式的方阵：

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

方阵A的逆矩阵 A^{-1} 满足条件：

$$A^{-1}A = AA^{-1} = I$$

逆矩阵 A^{-1} 表示由矩阵 A 所表示的线性变换的逆变换。

行列式是定义在方阵上的一个标量函数，矩阵 A 的行列式记为 $\det(A)$ 或 $|A|$ 。计算行列式的公式，可以用如下递归形式进行定义：

$$\begin{aligned} |a_{1,1}| &= a_{1,1} \\ \begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} &= a_{1,1}a_{2,2} - a_{2,1}a_{1,2} \\ \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} &= a_{1,1} \begin{vmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{vmatrix} - a_{1,2} \begin{vmatrix} a_{2,1} & a_{2,3} \\ a_{3,1} & a_{3,3} \end{vmatrix} + a_{1,3} \begin{vmatrix} a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{vmatrix} \end{aligned}$$

例如，

$$\begin{vmatrix} 2 & 1 & -3 \\ 0 & -1 & 5 \\ 4 & -2 & 3 \end{vmatrix} = 2 \begin{vmatrix} -1 & 5 \\ -2 & 3 \end{vmatrix} - 1 \begin{vmatrix} 0 & 5 \\ 4 & 3 \end{vmatrix} + (-3) \begin{vmatrix} 0 & -1 \\ 4 & -2 \end{vmatrix} \\ = 2(-3 + 10) - (0 - 20) - 3(0 + 4) = 22$$

在 3×3 矩阵上使用的展开方法可以经过扩展，从而递归地定义一般方阵的行列式。

431 可以证明，如果 A 和 B 都是方阵，则 $\det(A^T) = \det(A)$ ， $\det(AB) = \det(A)\det(B)$ 。

两个向量 $v = (x_1, y_1, z_1)$ 和 $u = (x_2, y_2, z_2)$ 的叉积 (cross product) $v \times u$ (也称为外积或向量积) 是定义在 R^3 上的运算。利用行列式，叉积可以符号化地表示为

$$v \times u = \begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} = (y_1z_2 - y_2z_1)i - (x_1z_2 - x_2z_1)j + (x_1y_2 - x_2y_1)k$$

从几何意义上来说， $v \times u$ 是垂直于 v 和 u 的向量，它的长度 $\|v \times u\| = \|v\|\|u\|\sin\theta$ ，这个长度等于由向量 v 和向量 u 张成的平行四边形的面积。图A-11显示了叉积的几何意义。

叉积是生成垂直于其他向量的向量的有用工具，例如两个向量 $(0,1,2)$ 和 $(3,4,5)$ 的叉积为

$$\begin{vmatrix} i & j & k \\ 0 & 1 & 2 \\ 3 & 4 & 5 \end{vmatrix} = -3i + 6j - 3k = (-3, 6, -3)$$

通过内积容易验证， $(-3, 6, -3)$ 和 $(0, 1, 2)$ 及 $(3, 4, 5)$ 是垂直的：

$$(-3, 6, -3) \cdot (0, 1, 2) = 0 + 6 - 6 = 0$$

$$(-3, 6, -3) \cdot (3, 4, 5) = -9 + 24 - 15 = 0$$

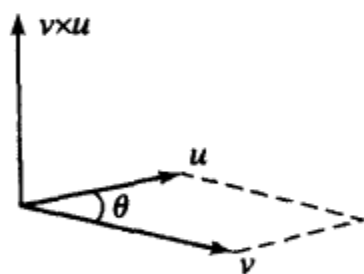
令 A 为一方阵，如果 v 是非零向量，而且存在某个数 λ 使得 $Av = \lambda v$ ，那么 v 就称为 A 的特征向量 (eigenvector)，而 λ 称为 A 的特征值 (eigenvalue)。

如果实矩阵 A 满足 $A^T = A$ ，则 A 是对称矩阵。如果实矩阵 U 满足以下条件，则 U 称为正交 (orthogonal) 矩阵：

$$U^T U = U U^T = I$$

谱定理 (spectral theorem) 保证，一个对称矩阵 A 可以分解成矩阵的积：

$$A = U \Lambda U^T$$



图A-11 叉积的几何解释

其中 U 是正交矩阵， Λ 是 A 的特征值所构成的对角矩阵。

更一般地，任何实矩阵可以分解成 $A = U\Lambda V^T$ ，其中 U 和 V 是正交矩阵而 Λ 是对角矩阵，这种分解称为奇异值分解（singular value decomposition, SVD）。

432

A.5 几何变换

A.5.1 齐次坐标

R^2 和 R^3 上的变换跟2D和3D图形直接相关， R^3 上的线性变换可以用 3×3 矩阵表示：

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = T \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

对计算机图形学有特殊意义的变换族，通常根据它们保持某些几何关系的性质来定义。投影变换（projective transformation）保持共线性，即它将直线映射成直线。仿射变换保持共线性和平行性，即它是一个将平行线映射成平行线的投影变换。上面定义的所有线性变换都是仿射变换，但并不是所有的仿射变换都是线性的。所有的仿射变换都是投影变换，但并不是所有的投影变换都是仿射变换。

由于投影变换和仿射变换并非总是线性的，它们不一定有上面所示的矩阵表示。但是，如果增加表示点的坐标系的维数，则对任何投影变换，都可以获得一种线性表示，这种坐标系称为齐次坐标（homogeneous coordinates）。

齐次坐标在标准坐标系的基础上增加了1D。例如，一个2D点可以用3个分量的齐次坐标 (x, y, w) 进行表示。同样地，一个3D点可以用四个分量的齐次坐标 (x, y, z, w) 进行表示。并且，只相差一个非零因数的齐次坐标表示的是相同的点。可见，空间中的点和齐次坐标之间的关系，并不是一对一的。

在2D空间中，齐次坐标用三个坐标代替两个坐标，来表示一个点： (x, y, w) 。

当 w 不为零时，该点的一般2D坐标 (X, Y) 可以按下式给出：

$$\begin{aligned} X &= x/w \\ Y &= y/w \end{aligned}$$

类似地，在3D空间中，齐次坐标用四个坐标代替三个坐标，来表示一个点： (x, y, z, w) ，或者写成列形式：

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

当 w 不为零时，该点的一般3D坐标 (X, Y, Z) 可以按下式给出：

$$\begin{aligned} X &= x/w \\ Y &= y/w \\ Z &= z/w \end{aligned}$$

在齐次坐标系中，四维向量用来表示3D点。根据定义，一个点的齐次坐标并不是唯一的，但任何非零数乘以四维齐次向量，仍然可以表示相同的3D点。

例如，齐次坐标 $(1, -2, 4, 1)$ 、 $(2, -4, 8, 2)$ 和 $(-3, 6, -12, -3)$ 都表示3D空间中同一点 $(1, -2, 4)$ 。

433

当w等于零时，没有通常意义的3D点与这个齐次坐标相关联。这种情况下，齐次坐标可以看做是无穷远点（points at “infinity”）的一种表示，这样的点可以看做是一种对方向的表示形式。例如，(1, 1, 0) 在2D空间中表示45度方向的无穷远点。

齐次坐标的一大优势在于，它使得所有的仿射变换和投影变换成为线性的，并且为这些变换提供了统一的矩阵表示形式。在3D空间中，变换可以表示成矩阵方程：

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

A.5.2 变换的分类

保持某些几何特性的变换，在几何学和图形学中有着重要的意义。
投影变换保持共线性，即它将直线映成直线。投影变换并不一定是线性的，例如，线性变换总把0向量映射成0向量，这对投影变换来说并不一定成立。但是，如上一节所述，利用齐次坐标，可以得到任意投影变换的线性表示。

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

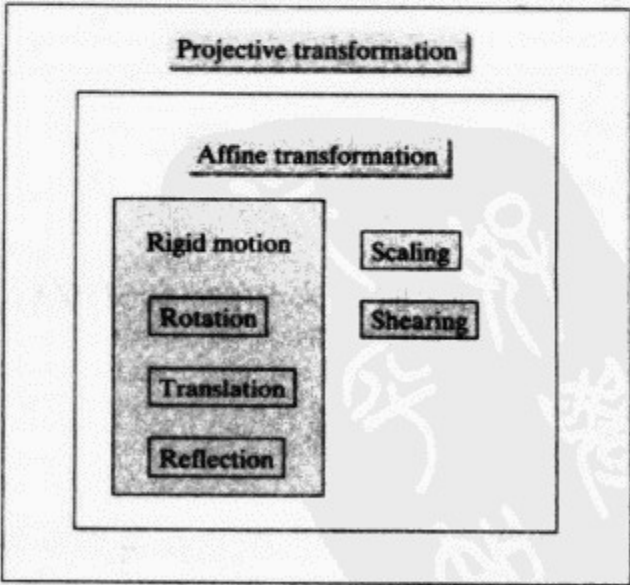
仿射变换是保持平行性的投影变换。例如，仿射变换将平行四边形映射成平行四边形。仿射变换是图形应用中最常用的变换，变换矩阵具有如下形式：

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

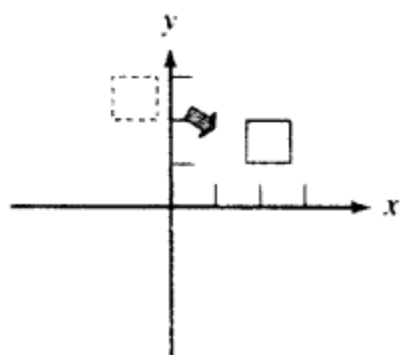
仿射变换的例子包括旋转变换、平移变换、缩放变换、反射变换和错切变换。
保持距离的仿射变换称为刚体运动（rigid motion）、欧几里德运动（Euclidean motion）或等距变换（isometry）。平移变换、旋转变换、反射变换都是刚体运动的例子。刚体运动的行列式不是1就是-1，反射变换的行列式为-1。

434

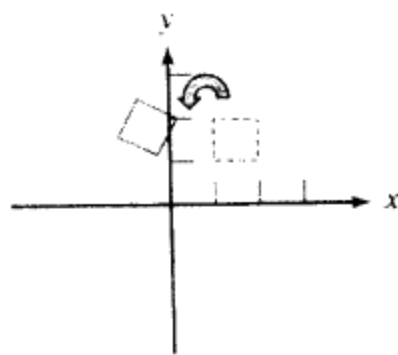
图A-12描述了常用几何变换的分类。
平移变换（translation）将一个空间的所有点移动一个固定量，它并没有改变任何对象的大小和方向，图A-13显示了一个2D平移的例子，这个平移在x和y方向上的移动量为 (2, -1)。
旋转变换将一个对象旋转。2D旋转变换是绕某个点进行的旋转变换，而3D旋转变换是绕某条直线进行的旋转变换。旋转变换并不改变尺寸，但是它会改变形体的方向，图A-14显示了关于原点的一个2D旋转变换。



图A-12 几何变换可以分为旋转变换、平移变换、缩放变换、反射变换和错切变换等



图A-13 平移变换将形体进行了平移

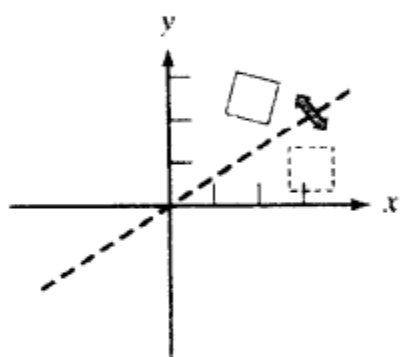


图A-14 旋转变换将形体进行了旋转

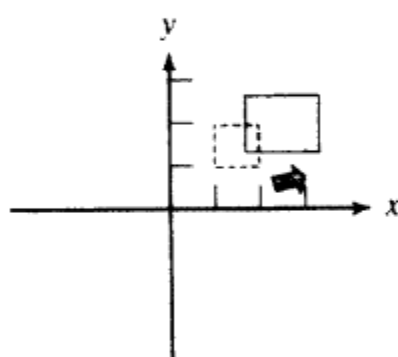
反射变换将对象映射成它的镜像，2D反射变换是关于某条直线进行的对称变换，而3D反射变换是关于某个平面进行的对称变换。对称保持距离的不变，但它改变了角度的方向。图A-15显示了一个2D反射变换的例子。

435

缩放变换通过在不同方向上乘以固定的因数来调整对象的大小，图A-16给出了一个2D缩放的例子。



图A-15 反射变换产生形体的镜像



图A-16 缩放调整了形体的大小

错切变换在某些方向上改变了对象。2D错切变换是沿着一条直线进行的变换而3D错切变换是沿着一个平面进行的变换，移动量与对象到固定直线或固定平面的带正负的距离成正比。图A-17给出了一个2D错切变换。

复数和四元数为变换提供了一种方便的解析表示形式，特别是对于旋转变换而言更是如此。在2D空间中，一个点可以和一个复数相关联，关于原点的2D旋转变换可以方便地表示为与复数 $e^{i\theta}$ 的乘法。

在3D空间中，一个点由一个纯四元数 $p = xi + yj + zk$ 所标识，关于原点的3D旋转变换可以表示成：

$$T_q(p) = qp\bar{q}$$

其中 q 是一个单位四元数，它可以写成

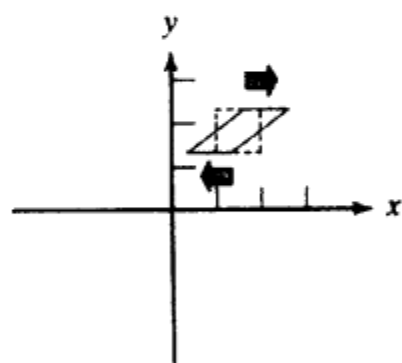
$$q = \cos \frac{\theta}{2} + u \sin \frac{\theta}{2}, \quad u = ai + bj + ck$$

单位纯四元数 u 定义了旋转轴， θ 是旋转的角度。

例如，如果需要定义一个旋转，其旋转角度为 $\pi/3$ ，旋转轴通过原点及点 $(1,1,1)$ ，则可以用以下四元数表示：

$$q = \cos \frac{\pi}{6} + \frac{1}{\sqrt{3}}(i + j + k) \sin \frac{\pi}{6} = \frac{\sqrt{3}}{2} + \frac{\sqrt{3}}{6}i + \frac{\sqrt{3}}{6}j + \frac{\sqrt{3}}{6}k$$

436



图A-17 错切变换偏移了形体

很明显,四元数表示形式提供了一种方便的方法,用于显式地指定旋转变换的旋转轴和旋转角度,以定义3D旋转变换。现在来考虑另一种情况。3D旋转变换的集合在复合变换下是封闭的,即两个旋转变换的组合结果仍然是一个旋转变换。但是,复合旋转变换的旋转轴和角度就不再是显而易见的了。如果首先绕y轴进行角度为 $\pi/2$ 的旋转变换,紧接着又绕x轴进行角度为 $\pi/2$ 的旋转变换,那么,最后形成的旋转轴和角度分别是多少?四元数表示形式为这类问题提供了一种简单的解决办法。两个旋转变换 T_{q_1} 和 T_{q_2} 的复合变换可以写成:

$$T_{q_1}(T_{q_2}(p)) = q_1 q_2 p \overline{q_2 q_1} = q_1 q_2 p \overline{q_1 q_2} = T_{q_1 q_2}(p)$$

因此,两个四元数的积表示两个旋转变换的复合变换。绕x轴进行角度为 $\pi/2$ 的旋转变换和绕y轴进行角度为 $\pi/2$ 的旋转变换,其四元数表示形式分别为:

$$q_1 = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i$$

$$q_2 = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j$$

那么复合旋转变换的四元数表示形式为:

$$\begin{aligned} q_3 &= q_1 q_2 = \left(\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i \right) \left(\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j \right) = \frac{1}{2}(1 + i + j + ij) \\ &= \frac{1}{2} + \left(\frac{i + j + k}{\sqrt{3}} \right) \frac{\sqrt{3}}{2} \\ &= \cos \frac{\pi}{3} + \left(\frac{i + j + k}{\sqrt{3}} \right) \sin \frac{\pi}{3} \end{aligned}$$

因此,最终的等价变换结果是进行一个旋转角度为 $2\pi/3$ 、旋转轴经过(0,0,0)和(1,1,1)的旋转变换。

A.6 微积分

给出一条由函数 $y = f(x)$ 表示的曲线,曲线上某点的切线(tangent line)斜率是函数的导数 $dy/dx = f'(x)$ 。为了得到过点 (x_0, y_0) 的切线方程,我们可以利用点斜式求出如下方程:

$$y - f(x_0) = f'(x_0)(x - x_0)$$

在3D空间中,曲线也可以用参数方程来表示:

$$\begin{aligned} x &= f(t) \\ y &= g(t) \\ z &= h(t) \end{aligned}$$

在由 $t = t_0$ 定义的点 (x_0, y_0, z_0) 处,切线的方向由向量 $(f'(t_0), g'(t_0), h'(t_0))$ 表示。切线的方程为:

$$\begin{aligned} x &= x_0 + f'(t_0)(t - t_0) \\ y &= y_0 + g'(t_0)(t - t_0) \\ z &= z_0 + h'(t_0)(t - t_0) \end{aligned}$$

一个光滑曲面在一给定点处有一切平面,切平面的法向量称为曲面在该点的曲面法线(surface normal),给定曲面的参数方程为:

$$x = f(u, v)$$

$$y = g(u, v)$$

$$z = h(u, v)$$

我们可以通过偏导数 (partial derivative) 得到曲面法线。考虑由导数形成的如下两个向量:

$$Du = (\partial x / \partial u, \partial y / \partial u, \partial z / \partial u) = (f_u, g_u, h_u)$$

$$Dv = (\partial x / \partial v, \partial y / \partial v, \partial z / \partial v) = (f_v, g_v, h_v)$$

曲面法线可以由叉积 $n = Du \times Dv$ 得到。例如, 由以下参数方程定义的一个曲面:

$$x = 2uv - 1$$

$$y = v^2$$

$$z = u^2 - v^3$$

偏导数为:

$$Du = (2v, 0, 2u)$$

$$Dv = (2u, 2v, -3v^2)$$

则可得曲面法线:

$$(2v, 0, 2u) \times (2u, 2v, -3v^2) = (-4uv, 4u^2 + 6v^3, 4v^2)$$

在由参数值 $u = 1, v = 1$ 定义的点 $(1, 1, 0)$ 处, 曲面法向量为 $(-4, 10, 4)$ 。对于由隐式方程 $F(x, y, z) = 0$ 给出的曲面, 它的曲面法线可以用梯度 (gradient) 来建立:

$$\nabla F(x, y, z) = (\partial F / \partial x, \partial F / \partial y, \partial F / \partial z)$$

例如, 双曲抛物面有方程 $z = xy$ 或 $xy - z = 0$ 。它的梯度为:

$$\nabla F(x, y, z) = (\partial F / \partial x, \partial F / \partial y, \partial F / \partial z) = (y, x, -1)$$

因此, 在点 $(-1, 2, -2)$ 处的曲面法线的方向为 $(2, -1, -1)$ 。

438

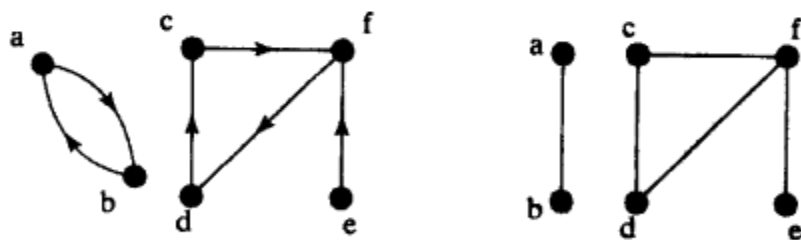
A.7 图论

图 (graph) 是一种数学结构, 它描述了一种称为顶点 (vertices, 也称作节点或点) 的对象集合上的关系。边 (edge, 也称作弧或线) 用来定义这种关系。一条边连接两个顶点, 它可以是无向的也可以是有向的。具有有向边的图称为有向图 (directed graph 或 digraph), 图A-18给出了一个有向图和一个无向图。

从形式化的角度, 无向图可以定义为 $G = (V, E)$, 其中 V 是顶点集而 E 是边集。连接顶点 u 和顶点 v 的边 uv 称为关联于顶点 u 和顶点 v 。如果两个顶点关联于同一条边, 那么就称这两个顶点是邻接 (adjacent) 的。路径 (path) 是不同的顶点和边交替出现的一个序列: $v_0 e_1 v_1 e_2 v_2 \cdots e_m v_m$, 其中边 e_i 和顶点 v_i, v_{i-1} 相关联。通常

我们可以忽略这个序列中边的定义, 把路径简单地定义为 $v_0 v_1 v_2 \cdots v_m$ 。环 (cycle) 类似于路径, 但它的起点和终点是重合的 $v_0 = v_m$ 。例如, 在图A-18所示的无向图中, $cdfe$ 是路径而 $cdfc$ 是环。

如果一个图的任意两个顶点都可以通过一条路径相连, 那么这个图是连通的 (connected)。如果一个图中没有环, 那么称它为无环的 (acyclic)。如果一个图连通且无环, 那么称它为树



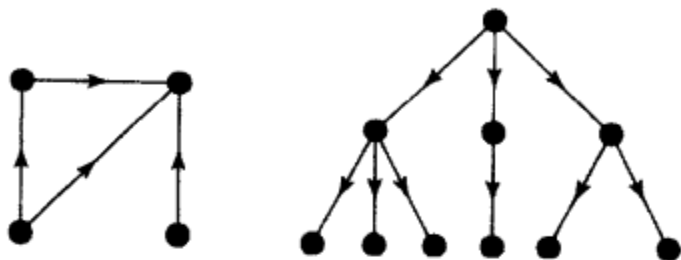
图A-18 一个有向图 (左侧) 和一个无向图 (右侧)

(tree)。树是具有很特殊性质的最小连通图。例如，树的边数和顶点数满足关系： $|E| = |V| - 1$ 。

有向图的相关概念可以类似地进行定义。例如，在图A-18所示的有向图中， $dcfe$ 是一条有向路径， $cfdc$ 是一个有向环。

DAG (directed acyclic graph, 有向无环图) 是没有有向环的有向图。但是，如果忽略DAG图中边的方向，产生的无向图可能会含有无向环，图A-19给出了这样的例子。

如图A-19所示，有根树 (rooted tree) 是一种全部由离开根节点的有向边所构成的树结构。有根树是DAG，但DAG并不一定是有根树。从树的根到任意顶点都存在唯一的路径。如果 uv 是一条从 u 到 v 的有向边，则我们



图A-19 一个DAG (左) 和一棵有根树 (右)

称 v 为 u 的子节点 (child) 及 u 是 v 的父节点 (parent)。没有子节点的顶点，称为叶节点 (leaf)。

关键术语

- 坐标系 (coordinate system) 将几何点与有序数元组的代数量相关联的方法。
- 参数方程 (parametric equation) 将坐标变量表达为参数函数的一组方程。
- 圆锥曲线 (conic section) 包括椭圆、抛物线和双曲线的一族曲线。
- 复数 (complex numbers) 由实数域扩展而来的一个数域。
- 四元数 (quaternion) 由复数域扩展而来的一个数域。
- 向量空间 (vector space) 定义了加法和数乘这两种运算的代数系统。
- 内积 (inner product) 两个向量的一个标量函数。
- 线性变换 (linear transformation) 保持线性组合关系的向量空间之间的映射。
- 矩阵 (matrix) 数的一个矩形阵列，通常用于表示线性变换。
- 行列式 (determinant) 方阵的一个标量函数。
- 叉积 (cross product) 在3D空间中定义的两个向量的向量值函数。
- 特征值 (eigenvalue)、特征向量 (eigenvector) 针对方阵，满足 $Av = \lambda v$ 的特殊值 λ 及向量 v 。
- 谱定理 (spectral theorem) 一个对称矩阵可以分解为： $A = U\Lambda U^T$ 。
- SVD (singular value decomposition, 奇异值分解) 一个矩阵可以分解为 $A = U\Lambda V^T$ 。
- 齐次坐标 (homogeneous coordinate) 增加另外一个维数来表示点的坐标系，从而使得所有的投影变换都可以线性地表示。
- 投影变换 (projective transformation) 保持共线性的几何变换。
- 仿射变换 (affine transformation) 保持平行性的几何变换。
- 刚体运动 (rigid motion) 保持距离的几何变换。
- 切线 (tangent line) 经过曲线上一点的直线，它的斜率与曲线在该点的变化率相等。
- 曲面法向量 (surface normal) 垂直于曲面上某点处的切平面的向量。
- 图 (graph) 由一组顶点和一组边构成的数学结构。
- 有向图 (digraph) 具有有向边的图。
- DAG 有向无环图。
- 树 (tree) 一个连通的无环图。

复习题

A.1 在2D坐标系中，画出以下点：

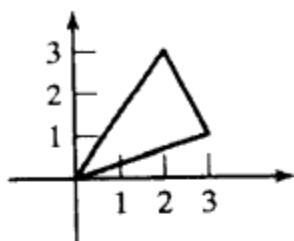
$(1,3)$, $(-2,1.5)$, $(0,-2)$, $(0,0)$

A.2 找出图A-20中三角形三个顶点的坐标。

A.3 求出两个3D点 $(2,1,3)$ 和 $(0,-2,5)$ 之间的距离。

440

- A.4 求出两个3D向量 $(2, 1, 3)$ 和 $(0, -2, 5)$ 的内积。
 A.5 求出两个3D向量 $(2, 1, 3)$ 和 $(0, -2, 5)$ 之间的夹角。
 A.6 求出两个3D向量 $(2, 1, 3)$ 和 $(0, -2, 5)$ 之间的叉积。
 A.7 求出经过点 $(0, 2)$ 、斜率为 -2 的直线的方程。
 A.8 求出直线 $-2x + 3y = 4$ 的斜率。
 A.9 求出以原点为圆心、半径为5的圆的方程。
 A.10 求出中心在 $(-1, 3)$ 、经过点 $(1, 3)$ 和点 $(-1, 4)$ 的椭圆的方程。
 A.11 求经过点 $(1, 3)$ 和 $(-1, 4)$ 的直线的参数方程。
 A.12 计算两个复数 $-2-5i$ 与 $1-3i$ 的和、差、积及商。
 A.13 求出复数 $-2-5i$ 的共轭复数及绝对值。
 A.14 求两个四元数 $-1-5i+k$, $1-3i-j+2k$ 的积。
 A.15 描述由四元数 $1/2-1/2i+1/2j+1/2k$ 定义的旋转。
 A.16 求绕轴为 $(1, -2, 2)$ 进行角度为 $2\pi/3$ 的3D旋转的四元数。
 A.17 求出下列矩阵的和与积



图A-20 习题A.2的三角形

$$\begin{bmatrix} 1 & 4 & -2 \\ 2 & 1 & 0 \\ -1 & -1 & -3 \end{bmatrix}, \begin{bmatrix} 2 & -1 & 3 \\ 0 & 7 & -2 \\ 1 & 5 & 4 \end{bmatrix}$$

- A.18 求下列矩阵的行列式:

$$\begin{bmatrix} -3 & 5 \\ 10 & -6 \end{bmatrix}$$

- A.19 求下列矩阵的行列式:

$$\begin{bmatrix} 1 & 4 & -2 \\ 2 & 1 & 0 \\ -1 & -1 & -3 \end{bmatrix}$$

- A.20 求 $\det(I)$ 。
 A.21 证明: $\det(A^{-1}) = 1/\det(A)$ 。
 A.22 证明下列矩阵是正交的:

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

- A.23 如果 A 是正交矩阵, 试证明: $\det(A) = \pm 1$ 。
 A.24 利用2D空间中的齐次坐标, 直线可以表示成方程: $ax + by + cw = 0$ 。如果两条直线平行, 求上述方程中系数的关系。
 A.25 考虑习题A.24中齐次坐标的直线方程。如果两直线平行, 是否可以通过解联立方程来求出一个交点?
 A.26 求出曲线 $y = x^3 + x - 1$ 在点 $(1, 1)$ 处的切线方程。
 A.27 求出下面3D曲线在点 $(1, -1, 0)$ 的切线方程:

$$\begin{aligned} x &= t + 1 \\ y &= t^2 - 1 \\ z &= t^3 \end{aligned}$$

- A.28 求下列曲线的曲面法向量:

$$\begin{aligned} x &= 2\cos u \cos v \\ y &= 3\sin u \cos v \\ z &= 4\sin v \end{aligned}$$

- A.29 如果一棵树有1000个顶点, 那么它有多少条边?
 A.30 如果一颗有根树有1000个叶节点, 并且每个非叶点都有2个子节点, 那么它有多少条边?

441

442

附录B 用AWT和Swing进行GUI编程

B.1 引言

目前在Java平台上，有AWT和Swing两套图形用户界面（GUI）编程接口，其中Swing比AWT新，也是目前推荐使用的接口。然而，Java 3D应用程序的绘制功能仍然依赖于AWT的Canvas 3D组件，因此Java3D程序中必定包含一些AWT组件。假定读者已经熟悉Swing编程，这里简要介绍AWT的框架结构及其与Swing的关系。

图形用户界面（graphical user interface, GUI）是现代计算机系统不可缺少的一部分。事实上，所有操作系统都对GUI提供了一定程度的支持，但是不同的GUI系统之间通常并不兼容。尽管不同平台上GUI编程的一些基本元素（如窗口系统和事件驱动编程）很相似，但是GUI编程通常非常依赖于本地系统的特性，因此可认为是与平台相关的。

Java语言从某种程度上可看做是为用户提供了一种平台无关的GUI编程环境。Java语言的标准API从一开始就包含了GUI编程接口。与Java的其他组件类似，Java的GUI库也是完全面向对象及平台无关的。

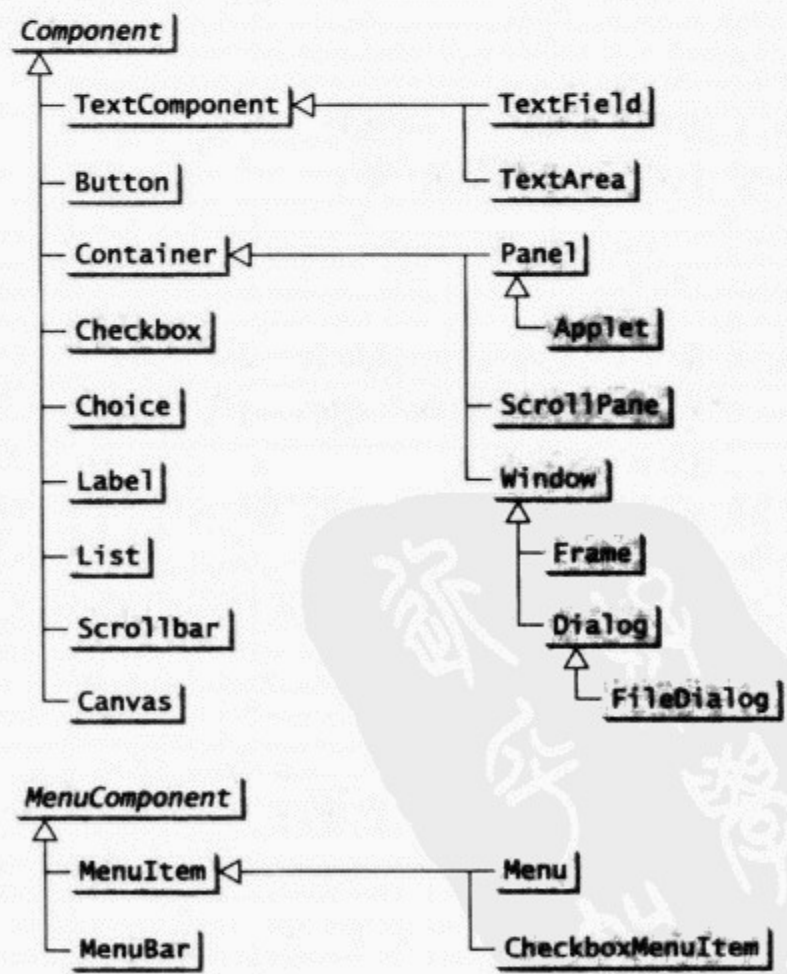
在Java的早期版本中，GUI功能由称为AWT（Abstract Window Toolkit，抽象窗口工具集）的API包实现。AWT定义了基本的事件驱动编程机制及一系列可视化组件。AWT组件相对比较简单，具有大部分的GUI系统所共有的一些特性，它们与本地系统中相似的组件建立直接映射实现。

Swing是一套新的GUI编程API库。它与AWT的主要不同在于，Swing重新定义了一套GUI组件。大部分的Swing组件不是重新包装的主机系统组件，它们完全是作为Java“轻量级”（lightweight）组件构建的，从而与本地GUI系统无关。如此一来，使Swing组件具有多种特性，它们的外观与任何主机平台无关。当然，Swing并没有完全取代AWT，它仅仅取代了AWT组件，Swing程序仍然使用了AWT的其他功能，例如布局管理器及事件处理。

B.2 AWT 组件

AWT组件是“重量级”（heavyweight）的组件。它们通常是不透明的，并且不允许事件（如鼠标点击事件）的传递。它们通过使用主机平台的组件来实现，因此，AWT组件在所有系统平台上只有有限的一组公共特性。

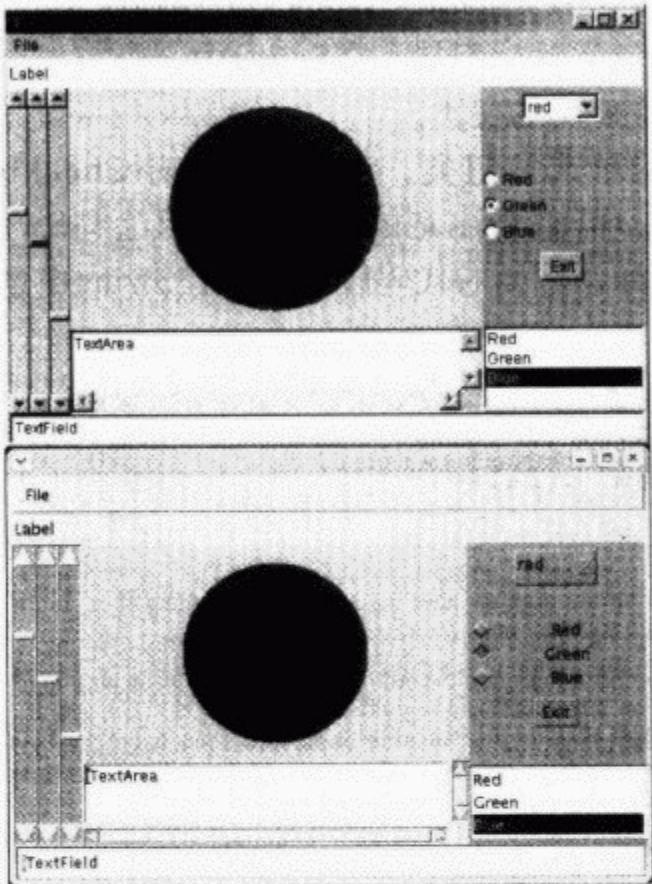
图B-1列出了主要AWT组件的类层次结构。



图B-1 主要的AWT组件

因为AWT组件是重量级的，所以一段AWT程序运行于不同的平台会有不同的外观效果。图B-2 显示了一段AWT程序在不同操作系统中运行的界面截图。尽管存在差别，但是一段GUI程序不经过任何修改，便可运行于不同的系统。

大部分AWT组件在Swing中有近似对等的组件。图B-3列出了Swing对AWT组件的替代方案。Swing版本组件通常以“J”为类名前缀，例如JTextFiled、JFrame及JPanel 分别与AWT组件 TextField、Frame和Panel相对应。



图B-2 AWT组件在不同平台下具有不同外观。上图：Window2000平台；下图：Linux平台

AWT Component	Swing Component
Frame	JFrame
Applet	JApplet
Button	JButton
Label	JLabel
TextField	JTextField
TextArea	JTextArea
Panel	JPanel
List	JList
Checkbox	JCheckBox, JRadioButton
Choice	JComboBox
Canvas	JPanel, JLabel
MenuBar	JMenuBar
Menu	JMenu
MenuItem	JMenuItem
CheckboxMenuItem	JCheckBoxMenuItem
Scrollbar	JSlider, JProgressBar
ScrollPane	JScrollPane
Dialog	JDialog, JOptionPane
FileDialog	JFileChooser

图B-3 AWT组件及对应的Swing组件

当然这也不是完全一一对应的。Swing组件的数量比AWT组件多，它们的设计思路有时也有所不同。

B.3 AWT编程

使用AWT组件编写的GUI程序与Swing程序有相同的基本结构。AWT中用户界面的构建、事件处理及界面绘制几乎与Swing相同。Swing则提供了比AWT更多的功能，因而使用纯AWT程序来模拟Swing程序的所有特性是困难的。然而，将一个Swing程序转换成相似的AWT程序，通常是可行的。如果我们要进行这种转换，或者根据已有的Swing知识编写AWT程序，可参考如下一些准则：

- 1) 将每一个Swing组件替换成最接近的AWT组件，可以使用图B-3来查找匹配的组件。
- 2) 与Swing中的JFrame和JApplet不同，AWT的顶层容器Frame和Applet中没有内容面板 (content pane)，组件将直接添加到Frame和Applet对象中。
- 3) Frame类中没有setDefaultCloseOperation方法，用户必须添加窗口事件监听器来实现上述功能。例如，下面的程序片段实现了当关闭窗口时，终止整个程序：

```
public static void main(String[] args){
    Frame frame = new Frame();
    //exit when the frame is closed
```

444
445


```

frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent ev){
        System.exit(0); //退出
    }
});
frame.setVisible(true); //设置窗口可见
}

```

4) Applet类没有菜单栏，而JApplet则有菜单栏。当然applet本身不一定需要菜单栏。

5) AWT组件没有paintComponent(Graphics)方法，而使用paint(Graphics)方法替代。

6) AWT编程中，用户定制的组件类通常由Canvas类派生而来，在Swing中，没有一个直接与之对应的类。在Swing中，JPanel通常可作为定制组件类的父类。

7) AWT中没有与Swing中JRadioButton对应的单选按钮类。用户可以使用Checkbox和CheckboxGroup对象来模拟实现单选按钮的功能。但需要注意的是，这种用法与Swing中JRadioButton 和ButtonGroup相结合的用法不同。例如，下面的代码段演示了在Swing中创建一组单选按钮的用法：

```

JRadioButton red = new JRadioButton("Red", true);
JRadioButton yellow = new JRadioButton("Yellow", false);
JRadioButton green = new JRadioButton("Green", false);
ButtonGroup group = new ButtonGroup();
group.add(red);
group.add(yellow);
group.add(green);

```

类似的AWT版本程序如下：

```

CheckboxGroup group = new CheckboxGroup();
Checkbox red = new Checkbox("Red", group, true);
Checkbox yellow = new Checkbox("Yellow", group, false);
Checkbox green = new Checkbox("Green", group, false);

```

8) AWT中Scrollbar组件通常用于近似替代JSlider组件，两者之间的主要不同点在于，Scrollbar触发AdjustmentEvent事件，而JSlider触发ChangeEvent事件。

程序清单B-1和B-2演示了如何使用AWT来重写Swing程序。程序清单B-1是一个典型的Swing程序，B-2则是功能类似的AWT程序（参见图B-4）。

程序清单B-1 SwingProg.java

```

1 //Swing程序
2
3 package appendixB;
4
5 import javax.swing.*;
6 import java.awt.*;
7 import java.awt.event.*;
8 import javax.swing.event.*;
9 //定义SwingProg类，继承自JFrame类，演示使用Swing组件编程
10 public class SwingProg extends JFrame
11     implements ActionListener, ChangeListener {
12
13     public static void main(String[] args) {
14         JFrame frame = new SwingProg(); //创建主窗口
15         frame.setVisible(true); //设置窗口可见
16     }
17

```



```
18 CirclePanel circle;
19 JSlider scrollR;
20 JSlider scrollG;
21 JSlider scrollB;
22
23 public SwingProg() {
24     setSize(500,350); //设置窗口大小
25     JMenuBar menuBar = new JMenuBar(); //创建菜单栏
26     setJMenuBar(menuBar); //设置窗口菜单栏
27     JMenu fileMenu = new JMenu("File");
28     menuBar.add(fileMenu); //添加一级菜单
29     JMenuItem exitItem = new JMenuItem("Exit");
30     exitItem.addActionListener(this); //设置事件监听器
31     fileMenu.add(exitItem); //添加二级菜单项
32     // 当窗口被关闭时退出程序
33     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34
35     Container cp = this.getContentPane();
36     cp.setLayout(new BorderLayout()); //设置布局器
37     // center布局
38     JPanel panel = new JPanel();
39     panel.setLayout(new BorderLayout());
40     cp.add(panel, BorderLayout.CENTER);
41     circle = new CirclePanel();
42     panel.add(circle, BorderLayout.CENTER);
43     JTextArea textArea = new JTextArea(3,10);
44     textArea.setText("TextArea");
45     panel.add(textArea, BorderLayout.SOUTH); //加入文本域控件
46     // north布局
47     JLabel label = new JLabel("Label");
48     cp.add(label, BorderLayout.NORTH);
49     // south布局
50     JTextField textField = new JTextField("TextField");
51     cp.add(textField, BorderLayout.SOUTH); //加入文本框控件
52     // west布局
53     panel = new JPanel();
54     panel.setLayout(new GridLayout(1,3));
55     cp.add(panel, BorderLayout.WEST);
56     scrollR = new JSlider(JSlider.VERTICAL, 0, 255, 0);
57     scrollR.addChangeListener(this);
58     panel.add(scrollR);
59     scrollG = new JSlider(JSlider.VERTICAL, 0, 255, 0);
60     scrollG.addChangeListener(this);
61     panel.add(scrollG);
62     scrollB = new JSlider(JSlider.VERTICAL, 0, 255, 0);
63     scrollB.addChangeListener(this);
64     panel.add(scrollB);
65     // east布局
66     panel = new JPanel();
67     panel.setBackground(Color.lightGray);
68     panel.setLayout(new GridLayout(4,1));
69     cp.add(panel, BorderLayout.EAST);
70     JPanel chPanel = new JPanel();
71     panel.add(chPanel);
```

```
72 JComboBox choice = new JComboBox();//创建JComboBox对象
73 choice.addItem("red");
74 choice.addItem("green");
75 choice.addItem("blue");
76 chPanel.add(choice);
77 JPanel cbPanel = new JPanel();
78 cbPanel.setLayout(new GridLayout(3,1));
79 panel.add(cbPanel);
80 ButtonGroup group = new ButtonGroup();
81 JRadioButton cbR = new JRadioButton("Red", true);//创建单选框
82 group.add(cbR);
83 cbPanel.add(cbR);
84 JRadioButton cbG = new JRadioButton("Green", false);
85 group.add(cbG);
86 cbPanel.add(cbG);
87 JRadioButton cbB = new JRadioButton("Blue", false);
88 group.add(cbB);
89 cbPanel.add(cbB);
90 JPanel btPanel = new JPanel();
91 panel.add(btPanel);
92 JButton button = new JButton("Exit");
93 button.addActionListener(this);
94 btPanel.add(button);
95 String[] listItems = {"Red", "Green", "Blue"};
96 JList list = new JList(listItems);
97 panel.add(list);
98 }
99 //ActionListener响应函数
100 public void actionPerformed(ActionEvent ev) {
101     String cmd = ev.getActionCommand();
102     if ("Exit".equals(cmd))
103         System.exit(0);
104 }
105 //ChangeListener响应函数
106 public void stateChanged(ChangeEvent ev) {
107     int r = scrollR.getValue();
108     int g = scrollG.getValue();
109     int b = scrollB.getValue();
110     circle.setColor(new Color(r, g, b));
111 }
112
113 }
114 //定义CirclePanel类, 继承自JPanel类
115 class CirclePanel extends JPanel {
116     private Color color = Color.black;
117
118     public CirclePanel() {
119         setBackground(new Color(220,220,220));
120     }
121     //重写组件绘制函数
122     public void paintComponent(Graphics g) {
123         super.paintComponent(g);
124         g.setColor(color);
125         int w = getWidth();
```



```
126     int h = getHeight();
127     int d = (w > h)? h : w;
128     d -= 30;
129     g.fillOval((w-d)/2,(h-d)/2, d, d);
130 }
131
132 public void setColor(Color c) {
133     color = c;
134     repaint();
135 }
136 }
```

程序清单B-2 AWTProg.java

```
1 //AWT程序
2
3 package appendixB;
4
5 import java.awt.*;
6 import java.awt.event.*;
7 //定义AWTProg类, 继承自Frame, 实现接口ActionListener和AdjustmentListener
8 public class AWTProg extends Frame
9     implements ActionListener, AdjustmentListener {
10
11     public static void main(String[] args) {
12         Frame frame = new AWTProg();//创建主窗口
13         frame.setVisible(true);//设置窗口可见
14     }
15
16     CircleCanvas circle;
17     Scrollbar scrollR;
18     Scrollbar scrollG;
19     Scrollbar scrollB;
20
21     public AWTProg() {
22         setSize(500,350);//设置主窗口大小
23         MenuBar menuBar = new MenuBar();//设置菜单栏
24         setMenuBar(menuBar);
25         Menu fileMenu = new Menu("File");
26         menuBar.add(fileMenu);
27         MenuItem exitItem = new MenuItem("Exit");
28         exitItem.addActionListener(this);
29         fileMenu.add(exitItem);
30         //关闭窗口时退出程序
31         addWindowListener(new WindowAdapter() {
32             public void windowClosing(WindowEvent ev){
33                 System.exit(0);
34             }
35         });
36         setLayout(new BorderLayout());//设置布局管理器
37         // center布局
38         Panel panel = new Panel();
39         panel.setLayout(new BorderLayout());
40         add(panel, BorderLayout.CENTER);
```

```
41 circle = new CircleCanvas();
42 panel.add(circle, BorderLayout.CENTER);
43 TextArea textArea = new TextArea(3,10);
44 textArea.setText("TextArea");
45 panel.add(textArea, BorderLayout.SOUTH);
46 // north布局
47 Label label = new Label("Label");
48 add(label, BorderLayout.NORTH);
49 // south布局
50 TextField textField = new TextField("TextField");
51 add(textField, BorderLayout.SOUTH);
52 // west布局
53 panel = new Panel();
54 panel.setLayout(new GridLayout(1,3));
55 add(panel, BorderLayout.WEST);
56 scrollR = new Scrollbar(Scrollbar.VERTICAL,0,1,0,255);
57 scrollR.addAdjustmentListener(this);
58 panel.add(scrollR);
59 scrollG = new Scrollbar(Scrollbar.VERTICAL,0,1,0,255);
60 scrollG.addAdjustmentListener(this);
61 panel.add(scrollG);
62 scrollB = new Scrollbar(Scrollbar.VERTICAL,0,1,0,255);
63 scrollB.addAdjustmentListener(this);
64 panel.add(scrollB);
65 // east布局
66 panel = new Panel();
67 panel.setBackground(Color.lightGray);
68 panel.setLayout(new GridLayout(4,1));
69 add(panel, BorderLayout.EAST);
70 Panel chPanel = new Panel();
71 panel.add(chPanel);
72 Choice choice = new Choice();
73 choice.add("red");
74 choice.add("green");
75 choice.add("blue");
76 chPanel.add(choice);
77 Panel cbPanel = new Panel();
78 cbPanel.setLayout(new GridLayout(3,1));
79 panel.add(cbPanel);
80 CheckboxGroup group = new CheckboxGroup();//添加CheckboxGroup对象
81 Checkbox cbR = new Checkbox("Red", group, true);
82 cbPanel.add(cbR);
83 Checkbox cbG = new Checkbox("Green", group, false);
84 cbPanel.add(cbG);
85 Checkbox cbB = new Checkbox("Blue", group, false);
86 cbPanel.add(cbB);
87 Panel btPanel = new Panel();
88 panel.add(btPanel);
89 Button button = new Button("Exit");
90 button.addActionListener(this);
91 btPanel.add(button);
92 List list = new List(3);
93 list.add("Red");
94 list.add("Green");
```

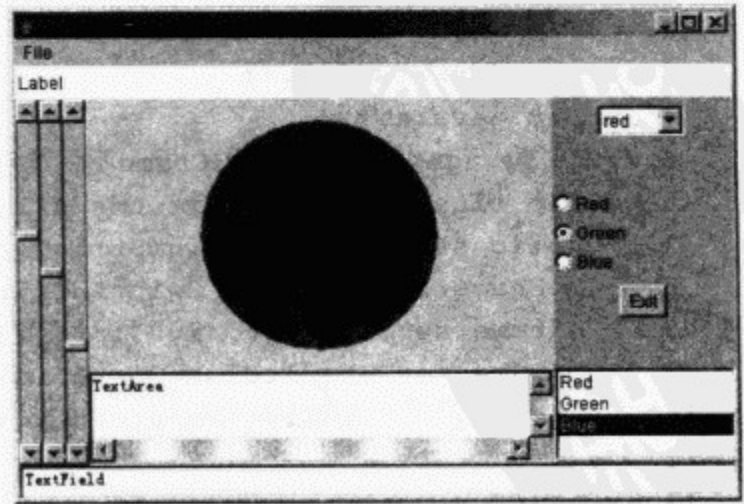
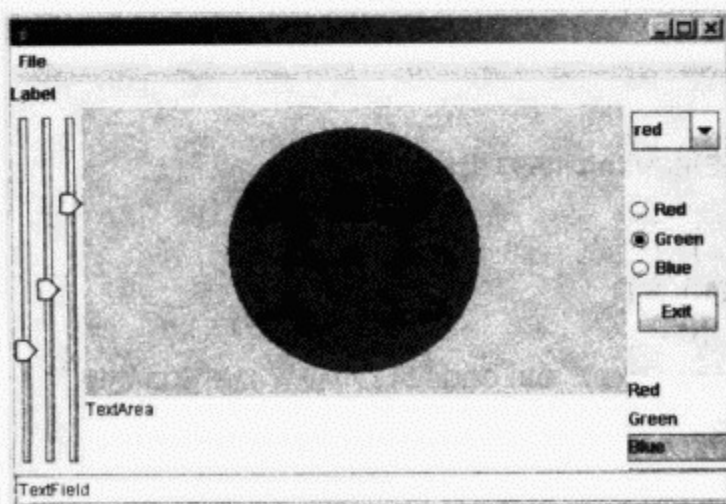


```

95     list.add("Blue");
96     panel.add(list);
97 }
98 //ActionEvent事件响应函数
99 public void actionPerformed(ActionEvent ev) {
100     String cmd = ev.getActionCommand();
101     if ("Exit".equals(cmd))
102         System.exit(0);
103 }
104 //AdjustmentEvent事件响应函数
105 public void adjustmentValueChanged(AdjustmentEvent ev) {
106     int r = scrollR.getValue();
107     int g = scrollG.getValue();
108     int b = scrollB.getValue();
109     circle.setColor(new Color(r, g, b));
110 }
111 }
112 //定义CircleCanvas类, 继承自Canvas类
113 class CircleCanvas extends Canvas {
114     private Color color = Color.black;
115
116     public CircleCanvas() {
117         setBackground(new Color(220,220,220));
118     }
119     //组件绘制方法
120     public void paint(Graphics g) {
121         super.paint(g);
122         g.setColor(color);
123         int w = getWidth();
124         int h = getHeight();
125         int d = (w > h)? h : w;
126         d -= 30;
127         g.fillOval((w-d)/2,(h-d)/2, d, d);
128     }
129
130     public void setColor(Color c) {
131         color = c;
132         repaint();
133     }
134 }

```

450



图B-4 Swing程序以及对应的AWT版本运行效果

451

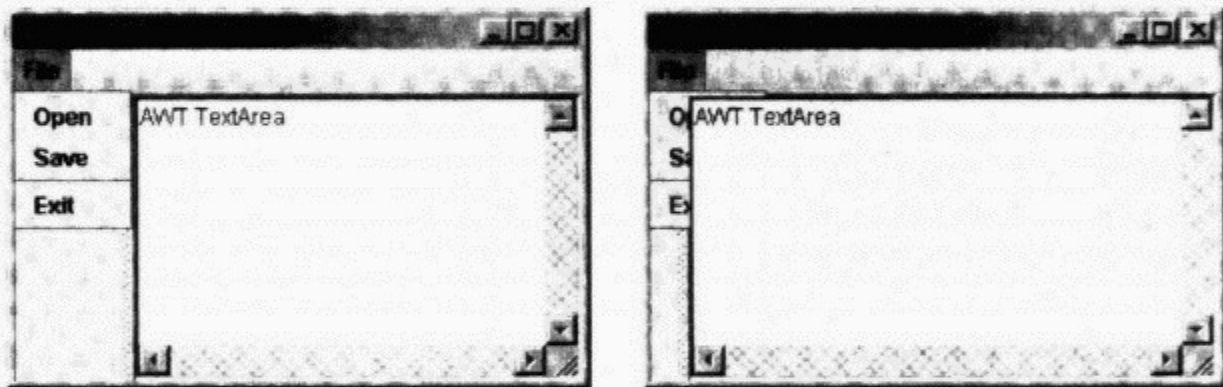
Swing版本中使用的组件有：JFrame、JPanel、JButton、JLabel、JTextField、JTextArea、JList、JComboBox、JRadioButton、JSlider、JMenuBar、JMenu及JMenuItem。相应地，在AWT版本中，这些组件由如下组件取代：Frame、Panel（和Canvas）、Button、Label、TextField、TextArea、List、Choice、Checkbox、Scrollbar、MenuBar、Menu及MenuItem。

在使用AWT中某些组件时，还必须考虑到如下的一些变化。AWT程序使用Frame对象作为直接包含其他组件的容器，而不是使用内容面板（content pane）。AWTProg类作为AdjustmentListener监听器来处理滚动条事件，而Swing中定义了ChangeListener来处理JSlider的相同事件。此外，在两个版本中，单选按钮的创建是不同的。

B.4 AWT和Swing混合编程

在同一个程序中，同时使用AWT和Swing是可行的。混合使用这两类组件并不会带来语法的错误，但是在最终生成的用户界面上，可能会有一些视觉效果偏差。

混合使用这些组件的最主要的问题，在于组件的z序（z-order）处理。z序决定了当有多个组件重叠时，哪一个组件将显示在最顶层。重量级的AWT组件是不透明的，它们的z序从主机系统得到。Swing组件可以是透明的，它们的z序是直接实现的。当一个AWT组件和一个Swing组件重叠时，AWT组件将会屏蔽Swing组件，这样可能造成错误的组件排列。例如，如下的程序包含了一个AWT组件（TextArea对象），该组件在一个JSplitPane组件的右半边。当选择菜单时，菜单和文本域组件将会重叠，此时菜单会被文本域组件屏蔽（如图B-5所示）。



图B-5 混合使用AWT和Swing组件可能会带来问题。
左图：当弹出的菜单与AWT文本域组件不重叠时，正常显示；
右图：由于z序的限制，弹出的菜单部分被AWT组件屏蔽

程序清单B-3 SwingAWT.java

```
1 package appendixB;
2
3 import javax.swing.*;
4 import java.awt.*;
5 //定义SwingAWT类，继承自JFrame类，演示混合使用Swing和AWT组件
6 public class SwingAWT extends JFrame {
7     public static void main(String[] args) {
8         JFrame frame = new SwingAWT();//创建主窗口
9         frame.setSize(300, 200);//设置窗口大小
10        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭窗口处理
11        frame.setVisible(true);//设置窗口可见
12    }
13
14    public SwingAWT() {
```



```
15    //设置菜单栏
16    JMenuBar mb = new JMenuBar();
17    setJMenuBar(mb);
18    JMenu menu = new JMenu("File");
19    mb.add(menu);
20    menu.add(new JMenuItem("Open"));
21    menu.add(new JMenuItem("Save"));
22    menu.addSeparator();//加入分隔符
23    menu.add(new JMenuItem("Exit"));
24    //添加控件
25    Container cp = this.getContentPane();//获得内容面板
26    cp.setLayout(new BorderLayout());//设置布局管理器
27    JSplitPane sp = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
28    JPanel panel = new JPanel();
29    sp.add(panel);//将panel对象放入sp
30    cp.add(sp);//将sp添加到内容面板
31    sp.add(new TextArea("AWT TextArea",10,10));// 向JSplitPane中加入文本域对象
32 }
33 }
```

当在同一个程序中必须同时使用AWT和Swing组件时，必须小心地避免z序问题。除了JMenuBar类以外，其他一些Swing组件，如JPopupMenu、JComboBox、JScrollPane及JInternalFrame等，也经常会出现类似的问题。在使用JPopupMenu和JComboBox对象时，我们可以选择调用如下方法，关闭轻量级的弹出事件：

```
setLightWeightPopupEnabled(false)
```

453

索引

索引中的页码为英文原书页码，与书中页边标注的页码一致

A

α -channel (α 通道), 81, 93

ActionListener interface (ActionListener接口), 123

actionPerformed method (actionPerformed方法), 121, 124

Affine transformations (仿射变换), 2, 70-78, 93

defined (定义), 70, 210

homogeneous coordinates (齐次坐标), 211

matrix form (矩阵形式), 211

rotation (旋转), 217-224

3D (三维), 210-224

AffineTransform class (AffineTransform类), 36, 38, 73, 92, 93, 123

AffineTransformOp class (AffineTransformOp类),
BufferedImageOp interface (BufferedImageOp接口), 110

Alpha class (Alpha类), 375

Alpha component (组件), color (颜色), 61

Alpha objects (Alpha对象), 150, 346

TestAlpha.java (TestAlpha.java源程序), 347-350
waveform of (波形), 346

Alpha parameters (Alpha参数), 346

alphaAtOneDuration parameter (alphaAtOneDuration参数), **Alpha** objects (Alpha对象), 346

alphaAtZeroDuration parameter (alphaAtZeroDuration参数), **Alpha** objects (Alpha对象), 346

AlphaComposite class (AlphaComposite类), 82, 84, 92

Ambient light (环境光), 282, 312

Ambient reflection (环境反射), 289, 313

AmbientLight class (AmbientLight类), 287, 311, 390

Analytic geometry (解析几何), 27

Anchor rectangle (锚点矩形), 65

Animation (动画), 3, 118-126, 131, 316, 345-378

billboards (公告板), 370-375

Clock2D.java (Clock2D.java源程序), 122-123

frame rate (帧速率), 118

geometric transformation and (几何变换), 210

implementing in Java (Java中的实现), 118-119

interaction compared to (与交互比较), 137

interpolators (插值器), 350-361

level of detail (LOD) (细节层次), 366-370

Life.java (Life.java源程序), 124-127

morphing (变形), 361-366

Rain.java (Rain.java源程序), 119-121

APIs (application programming interfaces, 应用程序接口), 8, 11, 15-17, 21

Appearance (外观), 197-204

ColorTetrahedron.java (ColorTetrahedron.java源程序), 199, 202-204

buttons (按钮), 203-204

defined (定义), 282

TestAppearance.java (TestAppearance.java源程序), 200-203

Appearance class (Appearance类), 149-150, 204, 290, 297

Applet class (Applet类), 140

Arc2D, 43

Arcs (弧), 49

Area class (Area类), 49-51

AreaGeometry.java (AreaGeometry.java源程序), 50-51

Arrow.java (Arrow.java源程序), 239-240, 243

Artificial shadows (人工阴影), 394

Ascent (上升量; 从Baseline到Glyph最上端的距离), 88

ASCII (ASCII编码), 87

Aspect ratio (长宽比), 251

Atmospheric attenuation (大气衰减), 294-297

TestFog.java (TestFog.java源程序), 295-297

Attenuation (衰减), 284

Attributes (属性), 197-204

AuralAttributes class (AuralAttributes类), 150

Avatar (替身), 272

AWTDemo.java (AWTDemo.java源程序), 17-19

Axes class (Axes类), 225

Axes object (Axes对象), 340

AxesBillboard.java (AxesBillboard.java源程序), 372-375

Axes.java (Axes.java源程序), 222-224

B

Back clip plane (后裁剪平面), 251

Background node (背景节点), 149, 160, 164

Backgrounds (背景), 154-160, 165

BackgroundSound class (BackgroundSound类), 390-391, 415

Baseline (基线), 88

BasicStroke class (BasicStroke类), 60, 67-68, 92, 93

Behavior, and picking (行为, 拾取), 334-339

Behavior class (Behavior类), 316-317

- class hierarchy (类层次结构), 316
- ClockBehavior.java** (ClockBehavior.java源程序), 321-323
- Clock.java** (Clock.java源程序), 319-321
- methods (方法), 316

Behavior object (Behavior对象), 316

Behavior-driven geometry update (行为驱动的几何更新), 405

Bernstein basis (Bernstein基), 98, 380

Bernstein polynomials (Bernstein多项式), 98, 380

Bézier curves (Bézier曲线), 98-100, 130-131, 380, 416

- mathematical definitions of (数学定义), 98

Bézier surface (Bézier曲面), 384-387

- BezierSurface.java** (BezierSurface.java源程序), 384-385
- Teapot.java** (Teapot.java源程序), 388-390
- TestBezierSurface.java** (TestBezierSurface.java源程序), 385-387

BezierCurve.java (BezierCurve.java源程序), 381-382, 384

BezierSurface.java (BezierSurface.java源程序), 384-385

Billboard class (Billboard类), 375

- behavior nodes (行为节点), 374
- constructors (构造函数), 370
- modes (模式), 370

Billboards (公告板), 370-375

- AxesBillboard.java** (AxesBillboard.java源程序), 372-375
- defined (定义), 370
- TestBillboard.java** (TestBillboard.java源程序), 371-372

Blue component, color (蓝色分量, 颜色), 61

Boolean operators (布尔运算符), 319

BoundingBox class (BoundingBox类), 156-157, 164

BoundingLeaf node (BoundingLeaf节点), 149, 155, 157, 164, 165

BoundingPolytope class (BoundingPolytope类), 156-157, 164

BoundingSphere class (BoundingSphere类), 156-157, 164

BoundingSphere object (BoundingSphere对象), 140

Bounds (作用范围边界), 154-160

- TestBounds.java** (TestBounds.java源程序), 158-160

Bounds class (Bounds类), 155-157, 164, 165, 297

Box object (Box对象), 195

Branch graphs, compiling (分支图, 编译), 165

BranchGroup node (BranchGroup节点), 146, 164, 336, 340

BranchGroup object (BranchGroup对象), 140, 145, 150-151, 272, 276, 277

B-spline curves (B样条曲线), 98-99, 131-132

BSpline.java (BSpline.java源程序), 101-103

BSplinePanel class (BSplinePanel类), 103

BufferedImage class (BufferedImage类), 108-109, 130-131, 306

BufferedImageOp interface (BufferedImageOp接口), 109, 130-131

BY_REFERENCE mode (BY_REFERENCE模式), 380, 399, 405, 416

C

C language (C程序设计语言), 11

Calendar class (Calendar类), 124, 130

Camera-based view model (基于摄像机的视图模型), 248

Canvas3D class (Canvas3D类), 405, 416

Canvas3D object (Canvas3D对象), 23, 140, 150-151, 160, 270, 276, 340

Capability bits (能力位), 160-161

capture method (capture方法), 409

Cellular automaton (细胞自动机), 124

ChangeBackground.java (ChangeBackground.java源程序), 161-163

Child node (子节点), 148-149

Children (孩子), root (根), 142

CIEXYZ color standard (CIEXYZ颜色标准), 60

Circle (圆周), 34

- in 2D graphics (在二维图形中), 34

Clip path (裁剪路径), 60, 93

ClipPanel class (ClipPanel类), 85

Clipping (裁剪), 85-87

TestClip.java (TestClip.java源程序), 85-86

Clipping path (裁剪路径), 85

Clock2D.java (Clock2D.java源程序), 122-123

- ClockBehavior.java** (ClockBehavior.java源程序), 321-323
- Clock.java** (Clock.java源程序), 319-321
- ClockPanel** class (ClockPanel类), 123-124
- CMYK (CMYK颜色模型), 60
- Coding schemes (编码方案), 87
- Color** class (Color类), 60-61, 92
 - TestColors.java** (TestColors.java源程序), 62-64
 - TestPaints.java** (TestPaints.java源程序), 65-67
- Color space (颜色空间), 60-67
- Color*** classes (Color为前缀的相关类), 171, 204
- ColorConvertOp** class (ColorConvertOp类), **BufferedImageOp** interface (BufferedImageOp接口), 110
- ColorCube** objects (ColorCube对象), 228
- Coloring (着色), 198
- Coloring rules (着色规则), 290
- ColoringAttributes** class (ColoringAttributes类), 149, 150, 161, 197-198, 204, 290-291
- ColorInterpolator** class (ColorInterpolator类), 351, 375
- ColorModel** (颜色模型), 108
- ColorPanel** class (ColorPanel类), 64
- Colors (颜色), 93, 197
- ColorTetrahedron.java** (ColorTetrahedron.java源程序), 199, 202-204
 - buttons (按钮), 203-204
- COMBINE** mode (COMBINE模式), 306
- Compatibility mode (兼容模式), 254-258
- CompatibilityMode.java** (CompatibilityMode.java源程序), 256-258
- Compiling, defined (编译, 定义), 160
- Compiling scene graphs (编译场景图), 160-163
- Composite transforms (复合变换), 229-233, 244
 - Mirror.java** (Mirror.java源程序), 230-232
- Compositing rules (复合规则), 93
 - transparency and (透明度), 81-85
- Compositing.java** (Compositing.java源程序), 82-84
- Composition (合成):
 - Composition.java** (Composition.java源程序), 79-80
 - defined (定义), 78
 - of transforms (变换), 78-80
- Composition rules (合成规则), 2-3
 - AlphaComposite** class (AlphaComposite类), 82, 84
 - Compositing.java** (Compositing.java源程序), 82-84
 - defined (定义), 81
 - Composition.java** (Composition.java源程序), 79-80
- Computer graphics (计算机图形学):
 - applications (应用), 3
 - defined (定义), 2
 - implementation details and (实现细节), 3
- Computer graphics programming (计算机图形学编程):
 - at different levels (不同层次), 3
 - evolution of (发展), 3-5
- Computer graphics programming (计算机图形学):
 - Graphics Kernel System (GKS), 11
 - hardware level (硬件层), 4-8
 - operating-system level support (操作系统层支持), 8-11
- Computer vision (计算机视觉), 26
- com.sun.j3d.utils.geometry** package (Java包), 193
- Cone** class (Cone类), 266
- Cone** object (Cone对象), 195, 226
- ConeSound** class (ConeSound类), 390, 415
- Constructive area geometry (构造区域几何), 49-51
 - AreaGeometry.java** (AreaGeometry.java源程序), 50-51
- contains** method (contains方法), 104
- contourCounts** array (contourCounts数组), 185
- Control points (控制点), 380
- ConvolveOp** class (ConvolveOp类), **BufferedImageOp** interface (BufferedImageOp接口), 110
- Coordinate systems (坐标系), 32-33
 - 2D geometry and (二维几何), 33-35
- createAudioDevice()** method (createAudioDevice方法), 391, 415
- createBackground** method (createBackground方法), 333-334
- createContent** method (createContent方法), 257
- createGeometry** method (createGeometry方法), 306
- createObject** method (createObject方法), 336
- createSceneGraph** method (createSceneGraph方法), 140
- createShadow** method (createShadow方法), 398
- createShape** method (createShape方法), 294, 297
- createTextureAppearance** method (createTexture Appearance方法), 306, 415
- createTransformedShape** method (createTransformed Shape方法), 74, 77
- createView** method (createView方法), 257
- cross** method (cross方法), 183
- Cross product (叉乘, 外积), 183
- CubeTexture.java** (CubeTexture.java源程序), 308-311
- Cubic Bézier curves (三次Bézier曲线), 42, 98-100
- Cubic B-spline curves (三次B样条曲线), 98
- Cubic curves (三次曲线), 49, 98

CubicCurve2D, 42

Cup.java (Cup.java源程序), 303-306

Curves (曲线), 416

curveTo method (curveTo方法), **GeneralPath** class (GeneralPath类), 106, 130

Custom behavior (自定义行为), 319

Custom primitives (自定义基元), 104-107

Heart.java (Heart.java源程序), 105-106

TestHeart.java (TestHeart.java源程序), 107

CustomPath.java (CustomPath.java源程序), 53-54

Cylinder class (Cylinder类), 226

Cylinder object (Cylinder对象), 195

D

DAG (directed acyclic graph, 有向无环图), 142-143, 165

Data type suffixes (数据类型后缀), 171

Data visualization (数据可视化), 337-342

 computer graphics and (计算机图形学), 189

DataViewer.java (DataViewer.java源程序), 338-340

deCasteljau algorithm (deCasteljau算法), 380, 416

decreasingAlphaDuration parameter (decreasingAlphaDuration参数), **Alpha** objects (Alpha对象), 347, 361

decreasingAlphaRampDuration parameter (decreasingAlphaRampDuration参数), **Alpha** objects (Alpha对象), 347

Depth cueing (景深效果处理), 294

DepthComponent class (DepthComponent类), 150

Derived fonts (衍生字体), 88

Descent (从Baseline到Glyph最下端的距离), 88

Device coordinate system (设备坐标系), 35

Device space (设备空间), 34

Diffuse reflection (漫反射), 289, 313

Directed graph (有向图), 142

Directed tree (有向树), 142

Directional light (方向光), 283, 312

DirectionalLight class (DirectionalLight类), 287, 311

DistanceLOD class (DistanceLOD类), 366, 375

DominantHand, 271

draw method (draw方法), 55

Drawing mode (绘制模式), 198

drawOval method (drawOval方法), 36

draw(Shape) method (draw(Shape)方法), 36

Graphics2D, 61

DrawShapes.java (DrawShapes.java源程序), 44-48

E

Ellipses (椭圆), 34, 42, 49

in 2D graphics (在二维图形中), 34

Elliptic arc (椭圆弧), 43

Enable head tracking (启用头部跟踪), 272

Environmental nodes (环境节点), 155

Epicycloid (外摆线), 39

Equations, graphing (方程, 绘图), 38-40

Euclidean motions (欧几里得运动), 70

Euler angles (欧拉角), 218

Even-odd rule (奇偶规则), 52-53

ExponentialFog class (ExponentialFog类), 311, 313

extrudeShape method (extrudeShape方法), 234-235

Extrusion (拉伸), 234-235

EYE_LINEAR mode (EYE_LINEAR模式), 307, 311

F

Field of view (fov, 视野), 251

fill method (fill方法), 55

fill(Shape) method (fill(Shape)方法), 36

Flat shading (平面明暗处理), 197

Focal length (焦距), 251

Fog class (Fog类), 311, 313

Font class (Font类), 93

Font face name (字体名), 87

Font metrics (字体标度), 88

FontFun.java (FontFun.java源程序), 89-90

GlyphClip.java (GlyphClip.java源程序), 91-92

Font style (字体样式), 88

Font3D class (Font3D类), 150, 204, 205

FontExtrusion class (FontExtrusion类), 150, 204

FontPanel class (FontPanel类), 90

FontRenderContext object (FontRenderContext对象), 88, 90-92

Fonts (字体), 93, 196-197

 derived (衍生), 88

 logical (逻辑的), 87

FORTTRAN (FORTRAN语言), 11

binding (绑定), 11

Fractal images (分形图像), creating (创建), 115-118

Mandelbrot.java (Mandelbrot.java源程序), 116-117

Fractals, defined (分形, 定义), 116

Frame rate (帧速率), 118

Front clip plane (前裁剪平面), 251

Front-elevation projections (前视投影), 248

Frustum (平截体), 251

Function (功能), 38

G

GeneralPath class (GeneralPath类), 51-52, 55, 100, 104

- curveTo** method (curveTo方法), 130
- Geometric models (几何模型), 40-49
 - DrawShapes.java** (DrawShapes.java源程序), 44-48
 - shapes (形状), 40-43
- Geometric objects, in 2D graphics (几何对象, 二维图形中), 34
- Geometric transformations (几何变换), 2-3, 209-246
 - composite transforms (复合变换), 229-233
 - extrusion (拉伸), 234-235
 - geometric primitives (几何基元), 233-234
 - orthogonal projection (正交投影), 210
 - rotation (旋转), 235-238
- Geometry (几何), 172-184, 205
 - Geometry** class (Geometry类), 173, 204, 290
 - class hierarchy (类层次结构), 173
 - GeometryArray** classes (GeometryArray类), 174-177, 204, 205
 - LineArray** class (LineArray类), 173-175
 - PointArray** class (PointArray类), 174
 - QuadArray** class (QuadArray类), 176-177
 - TriangleArray** class (TriangleArray类), 175-176
 - GeometryStripArray** classes (GeometryStripArray类), 177-178
 - LineStripArray** class (LineStripArray类), 177
 - TriangleFanArray** class (TriangleFanArray类), 178
 - TriangleStripArray** class (TriangleStripArray类), 177-178
 - IndexedGeometryArray** classes (IndexedGeometryArray类), 178-179, 204
 - IndexedLineStripArray** class (IndexedLineStripArray类), 179
 - IndexedTriangleFanArray** class (IndexedTriangleFanArray类), 179
 - IndexedTriangleStripArray** class (IndexedTriangleStripArray类), 179
 - normals (法线), 183-184
 - TestTetrahedron.java** (TestTetrahedron.java源程序), 180-182
 - Tetrahedron.java** (Tetrahedron.java源程序), 179-180, 182
- Geometry change (几何变化), 399-405
 - MovingShadow.java** (MovingShadow.java源程序), 400-404
- Geometry** class (Geometry类), 149-150
- GeometryArray** classes (GeometryArray类), 174-177, 185, 302, 361, 366, 399
- LineArray** class (LineArray类), 174-175
- PointArray** class (PointArray类), 174
- QuadArray** class (QuadArray类), 176-177
- TriangleArray** class (TriangleArray类), 175-176
- GeometryInfo** class (GeometryInfo类), 184-192, 204, 205
 - constructing geometry using (构造几何体使用), 185
- Dodecahedron.java** (Dodecahedron.java源程序), 186
- NormalGenerator** class (NormalGenerator类), 185
- polygon meshes, creating (多边形网格, 创建), 189-192
- TestDodecahedron.java** (TestDodecahedron.java源程序), 187-188
- ViewData.java** (ViewData.java源程序), 190-192
- GeometryStripArray** classes (GeometryStripArray类), 177-178
- LineStripArray** class (LineStripArray类), 178
- TriangleFanStripArray** class (TriangleFanStripArray类), 177-178
- TriangleStripArray** class (TriangleStripArray类), 177
- GeometryUpdater** interface (ViewData.java接口), 400, 405, 415, 416
- getGraphics()** method (getGraphics方法), 119
- getLineMetrics** method (getLineMetrics方法), 88, 90
- getOffScreenBuffer()** method (getOffScreenBuffer方法), 415
- getOutline** methods (getOutline方法), 92
- getPathIterator** method (getPathIterator方法), 104
- getRaster** method (getRaster方法), 115
- getReflection** method (getReflection方法), 232-233
- getStringBounds** method (getStringBounds方法), 88, 90
- GKS (Graphics Kernel System), 11
- GKS-3D, 11
- GL (Graphics Library) of Silicon Graphics Inc., 13
- GLCanvas**, 9-20
- GLJPanel**, 19-20
- Glyph (字形轮廓), 87
- GlyphClip.java** (GlyphClip.java源程序), 91-92
- GlyphVector** class (GlyphVector类), 90-92, 93
- GMatrix** class (GMatrix类), 212
- Gouraud shading (Gouraud阴影), 197
- Gradient (梯度), 409
- Gradient paint (渐变涂色), 65
- GradientPaint** class (GradientPaint类), 60, 92, 93
- Graphics, basic (图形学, 基础), 31-58
- Graphics** class (Graphics类), 34-35
- methods (方法), 36

Graphics contents (图形内容), 169-208
Graphics Kernel System (GKS), 11-13
Graphics objects (图形对象), 2
Graphics system (图形系统), components of (组件), 2
Graphics2D class (Graphics2D类), 35, 37, 64
 Hello2D.java (Hello2D.java源程序), 37-38
 methods (方法), 36
 Spirograph.java (Spirograph.java源程序), 39-40
Graphics-system layers (图形系统层次), 16
Graphing equations (绘图方程), 38-40
Green component (绿色分量), color (颜色), 61
Group class (Group类), 164
Group nodes (Group节点), 145-148
GUI (graphical user interface, 图形用户界面), 11
fundamental principle of (基本原理), 3
popularity of (流行), 3
GullCG class (GullCG类), 336

H

Hardware devices (硬件设备), 3
Hardware level, computer graphics programming (硬件层, 计算机图形编程), 4-8
Head tracking (头部跟踪), 271-275, 277
 6DOF (six-degrees-of-freedom) tracking devices (6自由度跟踪设备), 271
 enabling (启用), 272
 HeadTracking.java (HeadTracking.java源程序), 272-274
 LineAxes.java (LineAxes.java源程序), 274-275
Head-tracking information (头部跟踪信息), 248
HeadTracking.java (HeadTracking.java源程序), 272-274
Heart.java (Heart.java源程序), 105-106
Hello2D.java (Hello2D.java源程序), 37, 37-38
Hello3Dbackground.java (Hello3Dbackground.java源程序), 154-155
Hello3DfullGraph.java (Hello3DfullGraph.java源程序), 152-154
Hello3D.java (Hello3D.java源程序), 138-140
HiResCoord objects (HiResCoord对象), 144, 164
Homogeneous coordinates (齐次坐标), 170, 211

I

Illumination models (光照模型), 288-289
Image class (Image类), 107
Image processing (图像处理), 26, 98, 107-115
 copy operation (复制操作), 115
 edge-detection operator (边缘检测算子), 115

fractal images (分形图像), creating (创建), 115-118
ImageProcessing.java (ImageProcessing.java源程序), 111-114
printing (打印), 127-131
rescale operation (重缩放操作), 115
rotation operation (旋转操作), 115
sharpen operator (锐化算子), 115
smooth operator (平滑算子), 114-115
system (系统), 108
Image sharpening (图像锐化), 114
ImageComponent2D class (ImageComponent2D类), 312
ImageComponent2D object (ImageComponent2D对象), 409
ImageComponent3D object (ImageComponent3D 对象), 409, 415
ImageConsumer interface (ImageConsumer接口), 107
ImageIO class (ImageIO类), 110-111
ImageObserver interface (ImageObserver接口), 107, 109
ImagePanel class (ImagePanel类), 114
ImageProducer interface (ImageProducer接口), 107
Immediate model (Immediate模型), 108
“Immediate-mode” system (“直接模式”系统), 2
Implicit equation (隐方程), 172
increasingAlphaDuration parameter (increasingAlpha Duration参数), **Alpha** objects (Alpha对象), 346, 361
IndexedGeometryArray classes (IndexedGeometry Array类), 178-179, 302
 IndexedLineStripArray class (IndexedLineStrip Array类), 179
 IndexedTriangleFanArray class (IndexedTriangle FanArray类), 179
 IndexedTriangleStripArray class (Indexed TriangleStripArray类), 179
initialize method (initialize方法), 316, 323
Inner subclasses (内部子类), 41
InputDevice interface (InputDevice接口), 271, 276
Interaction (交互), 3, 323-334
 compared to animation (与动画比较), 137
 defined (定义), 316, 323
 mouse behaviors (鼠标行为), 323-334
Interaction-related classes (交互相关类), 324
Internal node (内部节点), 142
interpolate() method (interpolate方法), **Point3d** class (Point3d类), 415
Interpolator class (Interpolator类), 346, 350, 375
Interpolators (插值器), 350-361, 376

- and **Alpha** objects (与Alpha对象), 350
- ColorInterpolator** class (ColorInterpolator类), 351
- KBSplinePathInterpolator** class (KBSplinePathInterpolator类), 352
- PathInterpolator** class (PathInterpolator类), 351-352, 375
- Pendulum.java** (Pendulum.java源程序), 358-361
- PositionPathInterpolator**, 351-352
- RotationInterpolator** class (RotationInterpolator类), 351, 361
- RotationPathInterpolator**, 352, 375
- ScaleInterpolator** class (ScaleInterpolator类), 351
- ScalePathInterpolator**, 352
- SwitchValueInterpolator** class (SwitchValueInterpolator类), 351
- TCBSplinePathInterpolator** class (TCBSplinePathInterpolator类), 352
- TestInterpolator.java** (TestInterpolator.java源程序), 352-356
 - and time (时间), 350
- TransformInterpolator** class (TransformInterpolator类), 351
- TransparencyInterpolator** class (TransparencyInterpolator类), 351
- Intersections (相交), 342
- intersects** method (intersects方法), 104
- Isometries (等距), 70
- iterCount** method (iterCount方法), 118
- J
- Java (Java程序设计语言), 16-26
 - Abstract Window Toolkit (AWT, 抽象窗口工具包), 17
 - AWTDemo** class (AWTDemo类), 19
 - AWTDemo.java** (AWTDemo.java源程序), 17-19
 - defined (定义), 16-17
 - Java 2D, 21-23
 - JOGLDemo.java** (JOGLDemo.java源程序), 19-20
 - OpenGL, 19-20
 - programming language (编程语言), 16-26
 - Swing package (Swing包), 17
- Java 2D, 21-23, *See also* 2D graphics (参见二维图形学)
- Demo2D.java** (Demo2D.java源程序), 21-22
- Graphics2D** class (Graphics2D类), 21
- Java 3D, 23-26, *See also* 3D graphics (参见三维图形学)
 - Canvas3D**, 23
 - Demo3D.java** (Demo3D.java源程序), 23-25
 - Hello3DfullGraph.java** (Hello3DfullGraph.java源程序), 152-154
 - installing (安装), 140-141
 - nodes (节点), 145-149
 - program structure (程序结构), 150-153, 165
 - scene graphs (场景图), 26, 141-143
 - superstructure (超结构), 144-145
 - transformation support (变换支持), 210
 - utility classes (工具类), 277
 - view modes (视图模式), 277
- Java 3D API (Java3D编程接口), 138-141, 165
 - Hello3D.java** (Hello3D.java源程序), 138-140
 - javax.media.j3d**, 140
- Java 3D lighting models (Java3D光照模型), 290-294
- Java 3D view (Java3D视图):
 - classes directly related to viewing (与视图直接相关类), 253
 - compatibility mode (兼容模式), 254-258
 - CompatibilityMode.java** (CompatibilityMode.java源程序), 256-258
 - configuring (配置), 253-254
 - model (模型), 253-265
- Java Advanced Image (JAI, Java高级图像处理), 108
- Java Runtime Environment (JRE, Java运行环境), 141
- Java Virtual Machine (Java虚拟机), 17
- JavaDraw2DPanel** class (JavaDraw2DPanel类), 48-49
- javax.media.j3d**, 140
- javax.vecmath** package (javax.vecmath包), 140, 170, 205, 211, 244
- JBuilder (JBuilder集成开发环境), 141
- JOGL, defined (JOGL开发包, 定义), 17, 20
- JOGLDemo.java** (JOGLDemo.java源程序), 19-20
- K
- KBSplinePathInterpolator** class (KBSplinePathInterpolator类), 352
- Kernel (核心), 110
- Key controls (关键点), 328
- KeyNavigatorBehavior** class (KeyNavigatorBehavior类), 328
 - key controls (关键控制) 328
- TestKeyBehavior.java** (TestKeyBehavior.java源程序), 328-330
- Knots (节点), 351
- L
- Language binding (语言绑定), 11
- Leading (行距), 88

- Leaf (叶), 142
- Leaf** class (Leaf类), 316
- Leaf nodes (叶节点), 148-149
- AlternateAppearance** node (AlternateAppearance节点), 149
 - Background** node (Background节点), 149, 160, 164
 - Behavior** class (Behavior类), 148
 - BoundingLeaf** node (BoundingLeaf节点), 149, 155, 164
 - Clip** node (Clip节点), 149
 - Fog** node (Fog节点), 148
 - Light** node (Light节点), 148
 - Link** node (Link节点), 149
 - ModelClip** node (ModelClip节点), 149
 - Morph** node (Morph节点), 148-149
 - Shape3D** node (Shape3D节点), 148-149, 170, 172, 204
 - Sound** node (Sound节点), 149
 - SoundScape** node (SoundScape节点), 149
 - ViewPlatform** node (ViewPlatform节点), 149-151
- Level of detail (LOD, 细节层次), 366-370, 376
- DistanceLOD** class (DistanceLOD类), 366, 375
 - TestLOD.java** (TestLOD.java源程序), 367-370
- Life.java** (Life.java源程序), 124-127
- LifePanel** object (LifePanel对象), 127
- Ligature (连体字), 87
- Light** class (Light类), 284, 311
- Light, color of (光照, 颜色), 60
- Lighting (光照):
- combining texture mapping and (结合纹理映射), 303-306
 - Cup.java** (Cup.java源程序), 303-306
 - texture-coordinates generation (纹理坐标生成), 306-311
- Lights (光源), 282-288
- ambient light (环境光源), 282
 - directional light (方向光源), 283
 - Java 3D lighting models (Java3D光照模型), 290-294
 - Light** class (Light类), 284
 - Lighting.java** (Lighting.java源程序), 291-294
 - point light (点光源), 283
 - spotlight (聚光光源), 283
 - TestLights.java** (TestLights.java源程序), 285-287
- Linear algebra (线性代数), 27
- LinearFog** class (LinearFog类), 311, 313
- LineArray** class (LineArray类), 174-175
- LineAttributes** class (LineAttributes类), 150, 198, 204, 205
- LineAxes.java** (LineAxes.java源程序), 274-275
- LineMetrics** object (LineMetrics对象), 90-91, 93
- Lines (线), 33, 49
- LineStripArray** class (LineStripArray类), 177
- lineTo** method, **GeneralPath** class (lineTo方法, GeneralPath类), 51-52
- Link** node (Link节点), 147
- Live scene graph (活动场景图), 152, 165
- Local coordinate system (本地坐标系), 34
- Locale** class (Locale类), 143-145, 150-151, 164, 277
- LOD, See Level of detail (LOD)
- Logical fonts (逻辑字体), 87
- Logo.java** (Logo.java源程序), 240-242
- LookupOp** class (LookupOp类), **BufferedImageOp** interface (BufferedImageOp接口), 110
- LoopCount** parameter (LoopCount参数), **Alpha** objects (Alpha对象), 346
- ## M
- MainFrame** class (MainFrame类), 140, 163
- Mandelbrot set, defining (Mandelbrot集, 定义), 116
- Mandelbrot.java** (Mandelbrot.java源程序), 116-117
- MandelbrotPanel** class (MandelbrotPanel类), 118
- Marble texture (大理石纹理), 409
- Marble.java** (Marble.java源程序), 412-415
- Material** class (Material类), 150, 205, 290-291, 312
- Mathematical equations (数学方程), graphing (图表), 38-40
- Matrix (矩阵), 72
- Matrix classes (矩阵类), 211
- Matrix4d** class (Matrix4d类), 210-212, 214, 224, 243
- Matrix4f** class (Matrix4f类), 211, 243
- MatrixPanel.java** (MatrixPanel.java源程序), 212-215
- buttons (按钮), 214-215
- MediaContainer** class (MediaContainer类), 391, 415
- MediaTracker**, 109
- Mirror.java** (Mirror.java源程序), 230-232
- Modeler (建模组件), 2
- Modeling coordinate system (建模坐标系), 34
- Morph** class (Morph类), 361, 375
- Morphing (变形), 361-366
- defined (定义), 361
 - example (实例), 365
 - MorphingBehavior.java** (MorphingBehavior.java源程序), 364-366
 - Morphing.java** (Morphing.java源程序), 362-364
 - MorphingBehavior** class (MorphingBehavior类), 366
- Mouse behaviors (鼠标行为), 323-334
- data visualization (数据可视化), 337-342
 - key behaviors (关键行为), 328-330

and picking (拾取), 334-337

ViewPlatformBehavior classes (ViewPlatform Behavior类), 331-335

MouseAdapter class (MouseAdapter类), 19

MouseBehavior classes (MouseBehavior类), 323-324

constructors of (构造函数), 324

MoveGlobe.java (MoveGlobe.java源程序), 325-328

mouseClicked method (mouseClicked方法), 19

MouseListener, 19, 160

MouseRotate, 324, 340

MouseTranslate, 324, 340

MouseZoom, 324, 340

MoveGlobe.java (MoveGlobe.java源程序), 325-328

moveTo method, **GeneralPath** class (moveTo方法, GeneralPath类), 51-52

MoveView.java (MoveView.java源程序), 331-333

MovingShadow.java (MovingShadow.java源程序), 400-404

scene graph (场景图), 404

MultipleViews.java (MultipleViews.java源程序), 262-265

N

Netbeans (Netbeans开发环境), 141

Node classes (Node类), 164

NodeComponent classes (NodeComponent类), 143, 149, 164, 346

Nodes (节点), 145-149, 165

BranchGroup node (BranchGroup节点), 140, 145, 146, 150-151, 164

components (组件), 149-150

group (群组), 145-148

leaf (叶), 148-149

Link node (Link节点), 147

NodeComponent classes (NodeComponent类), 149

OrderedGroup node (OrderedGroup节点), 146

Primitive node (Primitive节点), 147

SharedGroup node (SharedGroup节点), 147

Switch node (Switch节点), 147-148

TransformGroup node (TransformGroup节点), 148

Noncommunicative transformation composition (不可交换的变换复合), 78

NondominantHand, 271

Nonzero rule (非零规则), 52-53

NORMAL_MAP mode (NORMAL_MAP模式), 311

NormalGenerator class (NormalGenerator类), 185

Normalization (标准化), 183-184

normalize method (normalize方法), 183

Normalized B-spline blending functions (标准B样条混合函数), 99

Normals (法线), 183-184

O

Object coordinate system (对象坐标系), 34

Object space (对象空间), 32, 34

Object transformations (对象变换), 2, 32-33, 73-74

OBJECT_LINEAR mode (OBJECT_LINEAR模式), 307, 311

Object-oriented programming (OOP, 面向对象编程), 17

Off-screen **Canvas3D** (离屏Canvas3D), 405

Off-screen rendering (离屏绘制), 405-409

OffScreen.java (OffScreen.java源程序), 406-408

renderOffScreenBuffer method (renderOffScreen Buffer方法), 409

waitForOffScreenRendering method (waitFor OffScreenRendering方法), 409

OffScreen.java (OffScreen.java源程序), 406-408

OpenGL, 13-16, 277

GL, 13-14

GLU (OpenGL Utility Library), 13-14

gluLookAt function (gluLookAt函数), 16

GLUT (OpenGL Utility Toolkit), 13-15

glutCreateWindow function (glutCreateWindow函数), 15

glutDisplayFunc function (glutDisplayFunc函数), 15

glutInitDisplayMode function (glutInitDisplay Mode函数), 16

glutMainloop function (glutMainloop函数), 15

glutPostRedisplay function (glutPostRedisplay函数), 16

Java and, 19-20

libraries (库), 13

OpenGLCircle.c (OpenGLCircle.c源程序), 14-15

OpenGLSphere.c (OpenGLSphere.c源程序), 15-16

Operating-system level support (操作系统层支持), 8-11

OrbitBehavior class (OrbitBehavior类), 331-333

OrderedGroup node (OrderedGroup节点), 146

Origin (原点), 33

Orthogonal projection (正投影), 210

Orthographic projections (正交投影), 248

P

Paint interface (Paint接口), 60, 64, 92

paintComponent method (paintComponent方法), 35, 37,

- 64, 84, 86, 92, 124
- Paints (涂色), 93
- Parallel projections (平行投影), 248-249
- Parametric equation (参数方程), 34, 172
- Pascal (Pascal编程语言), 11
- Patches (片), 189
- PathInterpolator** class (PathInterpolator类), 351, 351-352, 375
- PathIterator**, 51-52, 104
- Pendulum.java** (Pendulum.java源程序), 358-361
- perlinNoise** method (perlinNoise方法), 415
- PerlinNoise.java** (PerlinNoise.java源程序), 410-412
- Perlin's coherent noise (Perlin相干噪声), 409
- Perlin's noise (Perlin噪声), 380
- Perspective projections (透视投影), 136, 248-250
- phaseDelayDuration** parameter, **Alpha** objects (phaseDelayDuration参数, Alpha对象), 346
- PHIGS (Programmer's Hierarchical Interactive Graphics System), 11
- Phong illumination model (Phong光照模型), 282, 289, 312
- PhysicalBody** object (PhysicalBody对象), 150-151, 276
- PhysicalEnvironment** object (PhysicalEnvironment对象), 150-151, 271, 276
- pickAll()** method (pickAll方法), 270
- Picking (拾取), 248, 265-271, 277
 - and behavior (与行为), 334-339
 - defined (定义), 265
 - demo (演示), 271
 - and mouse behaviors (与鼠标行为), 334-337
 - TestPickBehavior.java** (TestPickBehavior.java源程序), 335-337
 - pick methods (拾取方法), 266
 - pick shapes (拾取形体), 266
 - PickCanvas** subclass (PickCanvas子类), 267, 270, 276
 - Picking.java** (Picking.java源程序), 268-270
 - PickIntersection** class (PickIntersection类), 267
 - PickTool** class (PickTool类), 267
 - SceneGraphPath** object (SceneGraphPath对象), 266-267
 - utility classes (工具类), 267
- PickMouseBehavior** classes (PickMouseBehavior类), 334, 337
- PickRotateBehavior** class (PickRotateBehavior类), 334
- PickShape** classes (PickShape类), 266, 276
 - class hierarchy (类层次结构), 266
- PickTranslateBehavior** class (PickTranslateBehavior类), 334
- PickZoomBehavior** class (PickZoomBehavior类), 334
- Pixels (像素), 107
- Point light (点光), 283, 312
- Point*** classes (Point前缀类), 171, 204
- Point3d** class (Point3d类), 172
- PointArray** class (PointArray类), 174
- PointAttributes** class (PointAttributes类), 150, 198, 204, 205
- PointLight** class (PointLight类), 287, 312, 390
- Points (点), 170
- PointSound** class (PointSound类), 391, 415
- Polygon** class (Polygon类), 43
- Polygon meshes (多边形网格), 172
 - creating (创建), 189-192
- PolygonAttributes** class (PolygonAttributes类), 150, 204, 205
 - POLYGON_FILL**, 198, 203
 - POLYGON_LINE**, 198, 203
 - POLYGON_POINT**, 198, 203
- Polygons (多边形), 43, 49
- Porter-Duff rules (Porter-Duff规则), 60, 81-82, 84, 93
 - PositionInterpolator**, 375
 - PositionPathInterpolator**, 351-352, 375
 - Primitive** node (Primitive节点), 147, 204
- Primitive type (Primitive类型), 184
- Primitives (基元), 170, 193-196, 205
 - setting the size of (设置大小), 193
- TestPrimitives.java** (TestPrimitives.java源程序), 193-195
- Printing (打印), 127-131
 - PageFormat** object (PageFormat对象), 127-128
 - Printable** interface (Printable接口), 127, 130
 - PrinterJob** class (PrinterJob类), 127, 130
 - Printing.java** (Printing.java源程序), 128-130
 - PrintPanel** class (PrintPanel类), 130
- Probabilistic model (概率模型), 81
- Procedural texturing (过程化纹理), 409
- processStimulus** method (processStimulus方法), 316, 323, 366
- Projection matrix (投影矩阵), 251, 277
- Projections (投影), 248-250, 251
 - front-elevation (前视), 248
 - orthographic (正交), 248
 - parallel (平行), 248-249
 - perspective (透视), 248-250
 - side-elevation (侧视), 248

top-elevation (顶视), 248
 Projective transformations (投射变换), 2, 137
 Pure quaternion (纯四元数), 217
 Push model (推模型), 107, 109

Q

QuadArray class (QuadArray类), 176-177
QuadCurve2D, 41
 Quadratic curves (二次曲线), 49, 98
 defined (定义), 41
 Quadrilateral patch (四边片), 189
Quat* classes (Quat前缀类), 171
Quat4d class (Quat4d类), 218, 224, 243
Quat4f class (Quat4f类), 218, 243
 Quaternion (四元数), 217-218
quatToEuler, 218-219

R

Rain.java (Rain.java源程序), 119-121
Raster class (Raster类), 108, 115, 130
 Raster images (光栅图像), 26
 Rational B-spline curve (有理B样条曲线), 99
Rectangle2D class (Rectangle2D类), 42
 Rectangles (矩形), 42, 49
 Red component (红色成分), color (颜色), 61
 Reflection (反射), 71-72, 216-217
REFLECTION_MAP mode (REFLECTION_MAP模式), 311
 Renderer (绘制者), 2
 Rendering, features (绘制, 特征), 2-3
RenderingAttributes, 150
renderOffScreenBuffer method (renderOffScreenBuffer方法), 409, 415
repaint() method (repaint方法), 119
RescaleOp class (RescaleOp类), **BufferedImageOp** interface (BufferedImageOp接口), 110
 reset method (reset方法), 55
 “Retained-mode” system (“保留模式”系统), 2
 RGB, 60
 RGB model (RGB模型), 289
 Rigid motions (刚性运动), 70
 Root, tree (根, 树), 142
RotateView.java (RotateView.java源程序), 259-261, 415
 Rotating tetrahedron (旋转四面体), 199, 257
 Rotation (旋转), 71, 217-224, 235-238, 244
 Axes.java (Axes.java源程序), 222-224
 quatToEuler, 218-219
 TestTorus.java (TestTorus.java源程序), 237-238

TestTransform.java (TestTransform.java源程序), 219-221, 224
Torus.java (Torus.java源程序), 236-237
 Rotation matrix (旋转矩阵), 217
RotationInterpolator class (RotationInterpolator类), 195, 351, 361, 375
Rotation.java (Rotation.java源程序), 226-228
RotationPathInterpolator, 352, 375
RotPosPathInterpolator, 351, 357, 375
RotPosScalePathInterpolator, 351, 375
 Round rectangle (圆角矩形), 42
RoundRectangle2D, 42, 43
Runnable interface (Runnable接口), 119, 130-131

S

ScaleInterpolator class (ScaleInterpolator类), 351, 375
ScalePathInterpolator, 352
 Scaling (缩放), 71-72, 216
 Scaling matrix (缩放矩阵), 216
 Scene graphs (场景图), 136, 165
 class hierarchy (类层次结构), 144
 defined (定义), 142
 Java 3D, 141-143
 legend of (图例), 143
 live (活动), 152, 165
 nodes of (节点), 142
 parts of (部分), 143
 transformations in (变换), 224-228
Rotation.java (Rotation.java源程序), 226-228
 Scene graphs, compiling (场景图, 编译) 160-163
 ChangeBackground.java (ChangeBackground.java源程序), 161-163
SceneGraphObject class (SceneGraphObject类), 144, 161, 164
SceneGraphPath object (SceneGraphPath对象), 266-267
Screen3D class (Screen3D类), 253-254, 265, 276
Sensor object (Sensor对象), 271, 275, 276
set method (set方法), 216
setCapability method (setCapability方法), 161
setClip method (setClip方法), **Graphics2D** object (Graphics2D对象), 92, 93
setDefaultCloseOperation method (setDefaultCloseOperation方法), 140
setGenMode method (setGenMode方法), 311
setOffScreenBuffer() method (setOffScreenBuffer方法):

- Canvas3D** class (Canvas3D类), 415
 - setPolygonMode** method (setPolygonMode方法), 198
 - setScale** methods (setScale方法), 216
 - setTexCoordGeneration** method (setTexCoordGeneration方法), 311
 - setTransform** method (setTransform方法), 74, 93
 - setXORMode** method (setXORMode方法), 49
 - SHAD_FASTEST**, 198
 - SHAD_FLAT**, 197
 - SHAD_GOURAUD**, 197
 - SHAD_NICEST**, 198
 - Shading model (明暗模型), 197
 - ShadowBehavior** class (ShadowBehavior类), 405
 - Shadows (阴影), 380, 394-399, 416
 - artificial (人工), 394
 - createShadow** method (createShadow方法), 398
 - Shadow.java** (Shadow.java源程序) 395-398
 - ShadowUpdater** class (ShadowUpdater类), 405
 - Shape** interface (Shape接口), 51, 131
 - methods (方法), 104
 - Shape3D** node (Shape3D节点), 148-149, 170, 172, 204, 270
 - SharedGroup** node (SharedGroup节点), 147
 - Shear (错切), 217
 - Shearing (错切), 71
 - Shearing matrix (错切矩阵), 217
 - Side-elevation projections (侧视投影), 248
 - SimpleUniverse** class (SimpleUniverse类), 140, 145, 151-152, 164
 - creating your own view (建立自己的视图), 261-265
 - MultipleViews.java** (MultipleViews.java源程序), 262-265
 - view settings in (视图设置), 258-261
 - RotateView.java** (RotateView.java源程序), 259-261
 - sleep** method, **Thread** class (sleep方法, Thread类), 119, 130
 - Sound (声音), 390-394, 416
 - adding (添加), 391
 - ConeSound** class (ConeSound类), 390, 415
 - PointSound** class (PointSound类), 391, 415
 - Sound3D.java** (Sound3D.java源程序), 391-394
 - Sound** classes (Sound类), 415
 - class hierarchy (类层次结构), 390
 - compared to **Light** classes (与Light类相比较), 391
 - Space (空间), 34
 - Specification of a view (视图描述), 251-252
 - Specular reflection (镜面反射), 137, 289, 313
 - Sphere** object (Sphere对象), 195
 - SPHERE_MAP** mode (SPHERE_MAP模式), 307, 311
 - Spirograph (螺旋线), 39
 - Spirograph.java** (Spirograph.java源程序), 39-40
 - SpiroPanel** class (SpiroPanel类), 40
 - Spline curves (样条曲线), 98-104
 - B-spline curves (B样条曲线), 98-99, 131-132
 - BSpline.java** (BSpline.java源程序), 101-103
 - normalized B-spline blending functions (规格B样条绘制函数), 99
 - rational B-spline curve (有理B样条曲线), 99
 - uniform B-spline (均匀B样条曲线), 99
 - Spotlight (聚光光源), 283, 312
 - SpotLight** class (SpotLight类), 287, 312
 - sRGB (standard RGB, 标准RGB颜色空间模型), 61
 - startTime** parameter, **Alpha** objects (startTime参数, Alpha对象), 346
 - startView** method (startView方法), 409
 - stripCounts** array, for a **POLYGON_ARRAY** (stripCounts数组), 185
 - Stroke** interface (Stroke接口), 60, 92, 93
 - Strokes (笔划), 60, 67-70, 93
 - BasicStroke** class (BasicStroke类), 60, 67-68, 92, 93
 - defined (定义), 67
 - examples of (实例), 70
 - TestStrokes.java** (TestStrokes.java源程序), 68-69
 - subdivide** method (subdivide方法), 384
 - Surfaces (曲面), 384-390, 416
 - Bézier surface (Bézier曲面), 384-387
 - Switch** node (Switch节点), 147-148
 - SwitchValueInterpolator** class (SwitchValueInterpolator类), 351, 375
- T
- TCBSplinePathInterpolator** class (TCBSplinePathInterpolator类), 352
 - Teapot.java** (Teapot.java源程序), 388-390
 - TestAlpha.java** (TestAlpha.java源程序), 347-350
 - TestAppearance.java** (TestAppearance.java源程序), 200-203
 - TestBezierCurve.java** (TestBezierCurve.java源程序), 382-384
 - TestBezierSurface.java** (TestBezierSurface.java源程序), 385-387
 - TestBillboard.java** (TestBillboard.java源程序), 371-372

- TestBounds.java** (TestBounds.java源程序), 158-160
- TestClip.java** (TestClip.java源程序), 85-86
- TestColors.java** (TestColors.java源程序), 62-64
- TestFog.java** (TestFog.java源程序), 295-297
- TestHeart.java** (TestHeart.java源程序), 107
- TestInterpolator.java** (TestInterpolator.java源程序), 352-356
- TestLights.java** (TestLights.java源程序), 285-287
- TestLOD.java** (TestLOD.java源程序), 367-370
- TestMatrix.java** (TestMatrix.java源程序), 213-214
- TestPaints.java** (TestPaints.java源程序), 65-67
- TestPickBehavior.java** (TestPickBehavior.java源程序), 335-337
- TestPrimitives.java** (TestPrimitives.java源程序), 193-195
- TestStrokes.java** (TestStrokes.java源程序), 68-69
- TestTetrahedron.java** (TestTetrahedron.java源程序), 180-182
- TestTorus.java** (TestTorus.java源程序), 237-238
- TestTransform.java** (TestTransform.java源程序), 219-221, 224
- Tetrahedron.java** (Tetrahedron.java源程序), 179-180, 182
- TexCoord*** classes (TexCoord为前缀的类), 171
- TexCoord2f** class (TexCoord2f类), 302, 311
- TexCoord3f** class (TexCoord3f类), 302, 311, 415
- TexCoord4f** class (TexCoord4f类), 302, 311
- TexCoordGeneration** class (TexCoordGeneration类), 306-311, 150
- CubeTexture.java** (CubeTexture.java源程序), 308-311
- Text2D** class (Text2D类), 196-197, 204, 205
- Text3D** class (Text3D类), 140, 196-197, 204, 205
- Texts (文本), 60, 196-197
- Texture** class (Texture类), 150
- Texture coordinates (纹理坐标), 303-303
- Texture mapping (纹理映射), 150, 197, 198, 297-311, 313
 - 2D, creating (二维, 创建), 297-301
 - 3D (三维), 380
 - combining lighting and (结合光照), 303-306
 - texture coordinates (纹理坐标), 302-303
- Texture paint (纹理填充), 65
- Texture pattern (纹理模式), 65
- Texture2D** class (Texture2D类), 312
- Texture3D** class (Texture3D类), 415
- TextureAttributes** class (TextureAttribute类), 150, 306, 312
- TextureCoordGeneration** class (TextureCoordGeneration类), 312, 313
- Texture-coordinate types (纹理坐标类型), 312
- Texture-coordinates generation (纹理坐标生成), 306-311
- TextureCubeMap** class (TextureCubeMap类), 312
- TextureLoader** class (TextureLoader类), 312
- TexturePaint** class (TexturePaint类), 60, 65, 92, 93
- TextureUnitState** class (TextureUnitState类), 150
- Thread** class (Thread类, 线程类), 119, 130-131
 - sleep** method (sleep方法), 130
- Thread safe, defined (线程安全性, 定义), 119
- 3D affine transform (三维仿射变换), 137
- 3D affine transformations (三维仿射变换), 210-224, 226-228
 - transformation matrix (变换矩阵), 211-215
- 3D curves (三维曲线), 380-384
- BezierCurve.java** (BezierCurve.java源程序), 381-382, 384
- TestBezierCurve.java** (TestBezierCurve.java源程序), 382-384
- 3D graphics (三维图形), 32
 - affine transforms (仿射变换), 137
 - basic (基础), 135-168
 - basic building blocks for (基础构造块), 137
- Hello3Dbackground.java** (Hello3Dbackground.java源程序), 154-155
- Hello3DfullGraph.java** (Hello3DfullGraph.java源程序), 152-154
- Java 3D API (Java 3D 应用程序接口), 138-141
- model and view (模型和视图), 136
- rendering process (绘制过程), 136-138, 141
- system (系统), 136-137, 164
- view frustum (视平截体), 137
- 3D rotation (三维旋转), 244
- Axes.java** (Axes.java源程序), 222-224
- quatToEuler**, 218-219
- TestTorus.java** (TestTorus.java源程序), 237-238
- TestTransform.java** (TestTransform.java源程序), 219-221, 224
- Torus.java** (Torus.java源程序), 236-237
- 3D rotations (三维旋转), 244
- 3D texture mapping (三维纹理映射), 380, 416
- 3D textures (三维纹理), 409-415
- Marble.java** (Marble.java源程序), 412-414
- PerlinNoise.java** (PerlinNoise.java源程序), 410-412
- Perlin's coherent noise function (Perlin相干噪声函数), 409

- procedural texturing (过程纹理), 409
 - 3D view (三维视图), 3
 - Timer** class (Timer类), 121-122, 130-131
 - Top-elevation projections (顶视投影), 248
 - Torus (环面), 235-239
 - TestTorus.java** (TestTorus.java源程序), 237-238
 - Torus.java** (Torus.java源程序), 236-237
 - transform** method (变换方法), 74
 - Transform3D** class (Transform3D类), 215-217, 218, 233-234, 243
 - convenience methods (便捷方法), 215-216
 - methods for multiplying transformation matrices to form composite transforms (将变换矩阵相乘以形成复合变换的方法), 229
 - Transformation matrix (变换矩阵), 211-215
 - GMatrix** class (GMatrix类), 212
 - MatrixPanel.java** (MatrixPanel.java源程序), 212-215
 - battons(按钮), 214-215
 - TestMatrix.java** (TestMatrix.java源程序), 213-214
 - Transformations (变换), 60, 77-78, 137
 - Arrow.java** (Arrow.java源程序), 239-240
 - compositions of (合成), 78-80
 - defined (定义), 210
 - geometric (几何的), 2-3, 209-246
 - Logo.java** (Logo.java源程序), 240-242
 - in scene graphs (场景图中), 224-228
 - Rotation.java** (Rotation.java源程序), 226-228
 - shared branch (共享分支), 239-243
 - Transformations.java** (Transformations.java源程序), 74-77
 - TransformGroup** node (TransformGroup节点), 148, 150-151, 195, 226, 243, 244, 294
 - TransformGroup** objects (TransformGroup对象), 224-226, 336
 - TransformInterpolator** class (TransformInterpolator类), 351, 375
 - TransformPanel** class (TransformPanel类), 77-78
 - Translation (平移), 34, 70, 72
 - defined (定义), 34
 - as a viewing transformation (视图变换), 33
 - Translation matrix (平移矩阵), 216
 - Transparency (透明度, 透明性), 198
 - compositing rules and (合成规则), 81-85
 - TransparencyAttributes** class (TransparencyAttributes类), 149, 204, 205
 - TransparencyAttributes** node (TransparencyAttributes节点), 198
 - TransparencyInterpolator** class (TransparencyInterpolator类), 351, 375
 - Tree (树), 142
 - TriangleArray** class (TriangleArray类), 175-176, 399-400
 - TriangleFanArray** class (TriangleFanArray类), 178
 - TriangleStripArray** class (TriangleStripArray类), 177-178
 - triggerTime** parameter, **Alpha** objects (triggerTime参数, Alpha对象), 346
 - Tuple** * classes (Tuple前缀类), 171
 - Turbulence function (扰动函数), 409
 - 2D geometric objects (二维几何对象), 33
 - 2D geometry, coordinate systems and (二维几何学, 坐标系), 33-35
 - 2D graphics (二维图形):
 - advanced topics (高级图形学课题), 97-133
 - transformations involved in (变换), 32-33
 - 2D graphics system (二维图形系统), 32
 - components of (组件), 32
 - rendering details (绘制细节), 59-96
 - rendering process (绘制过程), 32-33
 - 2D rendering process (二维绘制过程), 32-33
 - components of (组件), 32
 - 2D viewing (二维观察), 2
 - 2D world space (二维世界空间), 32
- U
- Unicode (统一双字节字符编码), 87
 - Uniform B-spline (均匀B样条曲线), 99
 - updateData** method (updateData方法), 400
 - User coordinate system (用户坐标系), 34
 - UserHead**, 271
- V
- Vector classes (向量类), 171-172
 - Vector spaces (向量空间), 170
 - Vector*** classes (Vector为前缀的相关类), 171, 204
 - Vector3d** class (Vector3d类), 172, 183
 - Vector3f** class (Vector3f类), 183
 - Vectors (向量), 170
 - Vertex colors (顶点颜色), 197
 - View (视图), defined (定义), 2
 - View center (视图中心), 252
 - View** class (View类), 150-151, 276
 - View frustum (观察平截体), 137, 248

- View plane (视平面), 252
 - View plane normal (vpn, 视平面法线), 252
 - View plate (视窗), 251
 - View projection (视图投影), 248
 - View up direction(up) (视图向上方向), 252
 - ViewData.java** (ViewData.java源程序), 190-192
 - ViewerAvatar**, 272, 275
 - Viewing matrix (观察矩阵), 251, 252, 277
 - Viewing transformations (观察变换), 2, 32
 - ViewPlatform** node (ViewPlatform节点), 149-151, 276
 - ViewPlatformBehavior** classes (ViewPlatform Behavior类), 331-335
 - MoveView.java** (MoveView.java源程序), 331-333
 - Viewpoint (视点), 252
 - View-reference point (vrp, 视参考点), 252
 - Views (视图), 247-279
 - camera-based (基于摄像机的), 248
 - defined (定义) 248
 - head tracking (头部跟踪), 271-275
 - Java 3D view model (Java3D视图模型), 253-265
 - picking (拾取), 265-271
 - projections (投影), 248-250
 - specification of (描述), 251-252
 - VirtualInputDevice** class (VirtualInputDevice类), 275
 - VirtualUniverse** class (VirtualUniverse类), 35, 143-145, 150-151, 163
- W**
- waitForOffScreenRendering()** method (waitForOffScreenRendering方法), **Canvas3D** class (Canvas3D类), 409, 415
 - Wakeup conditions (唤醒条件), 317-318
 - WakeupAnd** class (WakeupAnd类), 319
 - WakeupAndOrOfOrs** class (WakeupAndOrOfOrs类), 319
 - WakeupCondition** class (WakeupCondition类):
 - class hierarchy (类层次结构), 317
 - defined (定义), 317
 - WakeupOnAWTEvent**, 318, 324
 - WakeupOr** class (WakeupOr类), 319
 - WakeupOrOfAnds** class (WakeupOrOfAnds类), 319
 - WIN32 API (WIN32应用程序接口), 8
 - Winding rules (缠绕规则), 52, 55
 - WindowListener** (窗口监听器), 19
 - World coordinate system (世界坐标系), 34
 - World space (世界空间), 2, 32, 34, 2D (二维), 32
 - WritableRaster** class (WritableRaster类), 115-116, 118, 130-131
- X**
- x-axis (x轴, 横轴), 33
 - x-coordinate (x坐标, 横坐标), 33
- Y**
- y-axis (y轴, 纵轴), 33
 - y-coordinate (y坐标, 纵坐标), 33
- waitForOffScreenRendering()** method (waitForOffScreenRendering方法), **Canvas3D** class (Canvas3D类), 409, 415